HZ BOOKS
华章IT

资深架构师多年工作经验结晶

包含Kafka源代码分析与内部的实现原理，以及外部的维护工具、客户端编程、与第三方集成方式，书中穿插了大量的图片，讲解细致、便于理解

大数据
技术丛书

Source Code Analysis and Application of Kafka

Kafka源码解析与实战

王亮◎编著

机械工业出版社
China Machine Press

内容简介

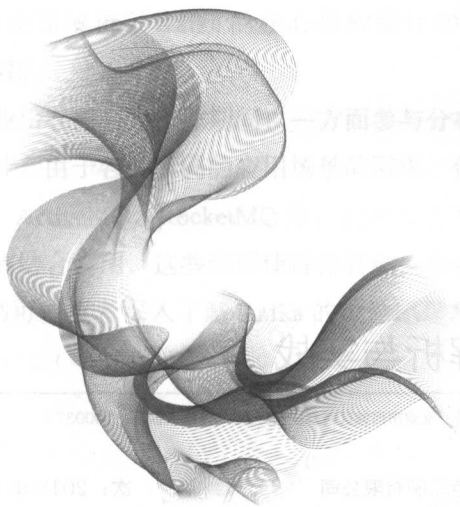
本书系统介绍Kafka的实现原理和应用方法，并介绍Kafka的运维工具、客户端编程方法和第三方集成方式，深入浅出、图文并茂、分析透彻。本书共10章，主要内容包括：第1章介绍Kafka诞生的背景和主要设计目标。第2章介绍Kafka的基本组成、拓扑结构以及内部的通信协议。第3章介绍Broker Server及内部的模块组成。第4章介绍Broker Server内部的九大基本模块。第5章介绍Broker的控制管理模块。第6章介绍Topic的管理工具。第7章从设计原则、示例代码、模块组成和发送模式四个方面介绍有关消息生产者的相关知识。第8章介绍两种消费者：简单消费者和高级消费者。第9章介绍Kafka的典型应用，包括与Storm、ELK、Hadoop、Spark典型大数据系统的集成。第10章介绍了一个综合实例，描述Kafka作为数据总线在安防整体解决方案中的作用。

大数据技术丛书

Source Code Analysis and Application of Kafka

Kafka源码解析与实战

王亮◎编著



机械工业出版社
China Machine Press

图书在版编目（CIP）数据

Kafka 源码解析与实战 / 王亮编著 . —北京：机械工业出版社，2017.10
（大数据技术丛书）

ISBN 978-7-111-58401-8

I. K… II. 王… III. 分布式操作系统 IV. TP316.4

中国版本图书馆 CIP 数据核字（2017）第 268376 号

Kafka 源码解析与实战

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号 邮政编码：100037）
责任编辑：吴 怡
印 刷：北京市荣盛彩色印刷有限公司 版 次：2018 年 1 月第 1 版第 1 次印刷
开 本：186mm × 240mm 1/16 印 张：17
书 号：ISBN 978-7-111-58401-8 定 价：79.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88379426 88361066 投稿热线：（010）88379604
购书热线：（010）68326294 88379649 68995259 读者信箱：hzit@hzbook.com

Preface 序

近些年来，大数据技术蓬勃发展，各种围绕大数据处理的平台技术，包括组件、工具、框架越来越丰富；相关的开源工具和实践资料也越来越多，其中消息队列便是一个重要的组成部分。对于一个大型系统而言，我们通常需要围绕消息来构建整个系统的逻辑，Kafka 便是目前最主流的消息系统之一。网络上有很多关于 Kafka 使用的文章，但是始终没有一本全面的从源码和设计上展开阐述的书籍。

值得庆幸的是，本书全面解析了 Kafka 的核心架构设计和源码，是国内少有的针对 Kafka 进行系统性讲解的书籍。

作者在浙江大华技术股份有限公司工作期间，一方面参与分布式数据库平台开发，一方面参与整体的系统架构设计。由于各种不同的应用场景的需求，作者所在公司内部用过多种不同消息队列，如 Kafka、ActiveMQ、RocketMQ 等，同时也实操了大量的 Hadoop、Spark 等大数据技术和消息队列的结合应用，这些经历使得作者能比较全面地从理论和实践两个视角去看待 Kafka。阅读本书可使读者深入了解 Kafka 的设计原理和使用技巧，相信读者一定会有所收获。

许焰

大华股份，研发副总经理

前　言 *Preface*

我开始接触分布式计算的时候，正好需要利用 Spark 结合 Kafka 进行流式处理。恰巧的是 Kafka 和 Spark 底层都是利用 Scala 语言编写的，并且当时市面上有关 Kafka 的中文书籍几乎没有，因此正好利用这个机会学习了 Scala 语言，并且通读了 Kafka 和 Spark 的源码，随后把日常的积累通过博客的形式慢慢记录下来。在这一年多的积累过程中，发现有关 Kafka 的中文书籍还是很缺乏，便有了总结出书的想法，而恰在这个时候吴怡编辑通过博客联系上了我，希望我把日常的积累总结成 Kafka 的专业性书籍，分享给更广大的从事大数据相关工作的人群。

本书将从初学者的角度出发，循序渐进地讲解 Kafka 内部的实现原理，但是由于 Kafka 是基于 Scala 语言编写的，因此为了更好地阅读本书，希望读者对于 Scala 语言有大致的了解。

阅读指南

本书将从 Kafka 的内部实现原理、运维工具、客户端编程以及实际应用这四个方面出发，系统阐述有关 Kafka 的各方面知识，全书共 10 章，每章的大致内容如下。

第 1 章介绍 Kafka 诞生的背景、Kafka 在 LinkedIn 内部的应用、Kafka 的主要设计目标以及为什么使用消息系统。

第 2 章介绍 Kafka 的基本组成、拓扑结构及其内部的通信协议。

第 3 章描述 Kafka 集群组成的基本元素 Broker Server 的启动以及内部的模块组成。通过阅读这一章，读者能对 Broker Server 有整体上的印象，为之后章节的阅读打下基础。

第 4 章描述 Broker Server 内部的九大基本模块：SocketServer、KafkaRequestHandlerPool、LogManager、ReplicaManager、OffsetManager、KafkaScheduler、KafkaApis、KafkaHealthcheck 和 TopicConfigManager。

第 5 章介绍 Broker Server 的控制管理模块 KafkaController，这个模块负责整个 Kafka 集群的管理，例如：Topic 的新建和删除、分区状态和副本状态的转换、集群的负载均衡管理等。

第 6 章介绍三个维护脚本：kafka-topics.sh、kafka-reassign-partitions.sh 和 kafka-preferred-replica-election.sh，它们分别涉及 Topic 的生命周期管理、Topic 分区的重分配和分区首选副本的选择。

第 7 章从设计原则、示例代码、模块组成和发送模式四个部分介绍有关消息生产者的相关知识，从设计原则至客户端编程，从客户端编程到内部实现原理，由浅入深，循序渐进地讲解。

第 8 章分别介绍两种消费者：简单消费者和高级消费者。针对每种消费者都将依次从设计原则、消费者流程、示例代码以及原理解析四个部分介绍消费者的相关知识。

第 9 章介绍 Kafka 与典型大数据系统的集成，包括：Kafka 和 Storm 的集成、Kafka 和 ELK 的集成、Kafka 和 Hadoop 的集成以及 Kafka 和 Spark 的集成。希望通过本章使读者对 Kafka 和第三方大数据平台集成有大致的了解。

第 10 章用综合实例描述了 Kafka 的应用，案例描述 Kafka 作为数据总线在安防整体解决方案中的作用，通过车辆人脸图片数据的入库、视频数据的入库、数据延时的监控、数据质量的监控、布控统计和容灾备份 6 个业务，简要阐述内部的实现原理。

本书是基于 0.8.2 版本的 Kafka 编写的，其相关配套的源码可以从 Kafka 的官方网站上下载，下载地址为 http://kafka.apache.org/downloads，也可以从开源或者私有软件项目托管平台 GitHub 上下载，下载地址为 https://github.com/apache/kafka。为了简化代码流程描述，笔者会将一些日志打印等不影响阅读的代码用“……”代替，如果需要知道“……”代表的实际含义，可以参考源码包中的真实代码。

本书特点

由浅入深，循序渐进：本书从 LinkedIn（领英）公司内部大数据架构讲起，引出消息队列 Kafka，接着讲解 Kafka 的基本架构，然后着重分析 Kafka 内部的各模块实现细节。从诞生背景至架构组成，再到内部实现细节，由浅入深，循序渐进，让读者在阅读时能够逐步了解 Kafka。

由里到外，层层剖析：本书不仅讲解 Kafka 内部的实现原理，而且还详细描述 Kafka 外部的维护工具，对外的客户端编程原理以及和第三方集成的方式。由里到外，层层剖析，让读者在阅读时能够更加全面地掌握 Kafka。

图文并茂，生动形象：本书在讲解 Kafka 的过程中穿插了大量的图片，直观地描述了工作原理，使读者在阅读时能够加深对代码的理解。

读者对象

本书适合以下人群阅读：

- ❑ 想熟悉典型消息系统架构的大数据从业人员。
- ❑ 想了解分布式系统开发的软件工程师。
- ❑ 想掌握 Kafka 内部实现原理的中高级开发人员。
- ❑ 想搭建传统大数据框架的系统分析师。

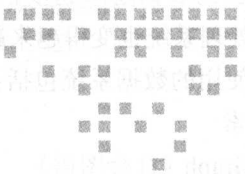
致谢

首先感谢我的夫人在我背后默默的付出，是她给了我动力，陪伴我度过了长达半年之久的枯燥时光，坚定了我完成此书的决心。其次感谢机械工业出版社吴怡编辑的鼓励和支持，是她促成了这本书的出版。接着感谢我的鱼儿们（布隆迪、金头虎、蓝茉莉、三间鼠和反游猫），每当我思绪混乱的时候可以静静地看着它们慢慢梳理。

在本书成书的过程中也得到了许多同事和同学的支持、鼓励，在此一并致谢。

由于作者水平及能力有限，加之时间仓促，本书难免存在错误和不妥之处，恳请广大读者批评指正，邮箱地址为：wangliang168219@126.com。

Contents 目　录



第 1 章 Chapter 1

Kafka 简介

Kafka 是一个高度可扩展的消息系统，它在 LinkedIn 的中央数据管道中扮演着十分重要的角色，因其可水平扩展和高吞吐率而被广泛使用，现在已被多家不同类型的公司作为多种类型的数据管道和消息系统。本章将聚集于 Kafka 诞生的背景、Kafka 在 LinkedIn 内部的应用、Kafka 的主要设计目标和为什么使用消息系统这四个方面，简单介绍典型的消息系统 Kafka，尤其在应用的时候需要多思考后两个方面：Kafka 的设计目标和应用层开发为什么需要使用消息系统。并以此为切入点，为之后的章节作铺垫。

1.1 Kafka 诞生的背景

对于一个高效的组织，所有数据需要对该组织的所有服务和系统是可用的，以便挖掘出数据的最大价值。数据采集和数据使用是一个金字塔的结构，底部为以某种统一的方式捕获数据，这些数据需要以统一的方式建模，以方便读取和处理。捕获数据的工作做扎实后，在这个基础上以不同方法处理这些数据就变得得心应手。

数据捕获的来源主要有两种：一种是记录正在发生的事件数据。比如 Web 系统中的用户活动日志（用户的点击选择等）、交警行业中的违章事件等。随着传统行业业务活动的数字化，事件数据正在不断增长，而且这个趋势没有停止。这种类型的事件数据记录了已经发生的事情，往往比传统数据库应用要大好几个数量级。因此对于数据的捕获、数据的处理提出了重大的挑战；另一种是经过二次分析处理之后的数据。对捕获的数据进行二次分析处理后得到的数据也需要记录保存，这里的处理指的是利用批处理、图分析等专有的数据处理系统进行了处理，这些加工后的数据可以作为数据捕获的第二个来源。

总之，捕获的数据越来越多，如何将这些巨量的数据以可靠的、完整的数据流方式传递给数据分析处理系统也变得越来越困难。

LinkedIn 使用的数据系统包括：

- 全文搜索
- Social Graph（社会图谱）
- Voldemort（键值存储）
- Espresso（文档存储）
- 推荐引擎
- OLAP（查询引擎）
- Hadoop
- Teradata（数据仓库）
- Ingraphs（监控图表和指标服务）

上述专用的分布式系统都需要经过数据源来获取数据，同时有些系统还会产生数据，作为其他系统的数据源。LinkedIn 曾尝试为每个数据源和目标构建自定义的数据加载，很显然这是不可行的。LinkedIn 有几十个数据系统和数据仓库。把这些系统和仓库联系起来，就会导致任意两两系统间构建自定义的管道，如图 1-1 所示。

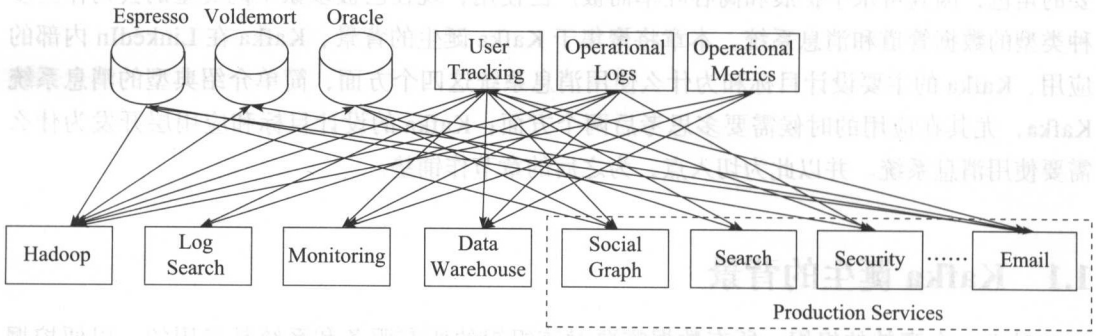


图 1-1　各系统之间的自定义管道

需要注意的是，数据是双向流动的，例如许多系统（数据库、Hadoop）同时是数据传输的来源和目的端。这就意味着我们最后要为那些系统建立两个通道：一个用于数据输入，一个用于数据输出。要避免上面的问题，我们需要如图 1-2 所示的通用方式。

我们需要尽量将每个生产者、消费者与数据源隔离。理想情况下，生产者或消费者应该只与一个数据源单独集成，这样就能访问到所有数据。根据这个思路想到增加一个新的数据系统：

- 作为数据来源或者数据目的地。
- 集成工作只需要连接这个新系统到一个单独的管道，而无须连接到每个数据的生产者和消费者。

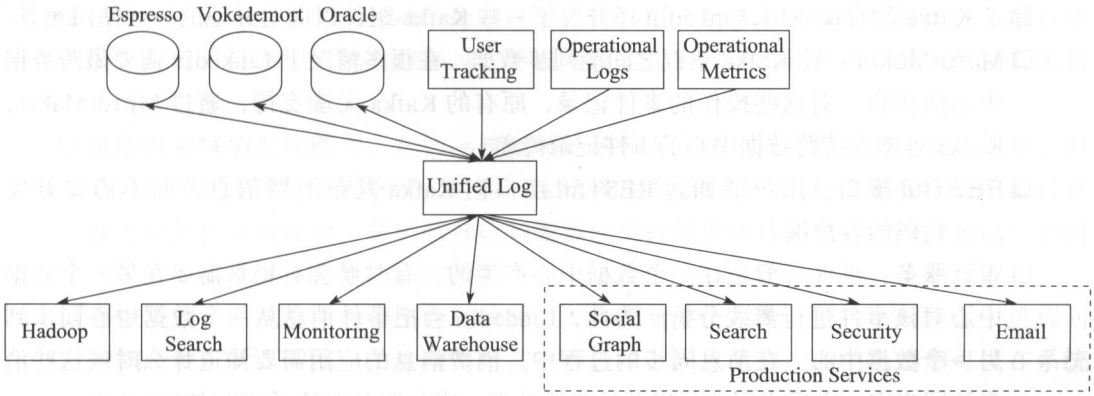


图 1-2　各系统之间的共享管道

这个新的数据系统就是 Kafka。Kafka 作为 LinkedIn 中的“中枢神经系统”，管理从各个应用程序汇聚到此的信息流，这些数据经过处理后再被分发到各处。Kafka 作为一个消息系统，进行消息的传递；同时它也是日志存储系统，以日志的形式存储了数据源的所有数据。

1.2　Kafka 在 LinkedIn 内部的应用

LinkedIn 的工程师团队已经把 Kafka 打造为管理信息流的开源解决方案。他们把 Kafka 作为消息中枢，帮助公司的各个应用以松耦合的方式在一起工作。LinkedIn 已经严重依赖于 Kafka，并且基于 Kafka 的生态系统，LinkedIn 开发出了一些开源组件和公司内部组件。

Kafka 在 LinkedIn 中的使用场景如下所述：

- **系统监控**：LinkedIn 内所有的主机都会往 Kafka 发送系统健康信息和运行信息，负责展示运维信息和报警的系统则从 Kafka 订阅获取这些运维信息，处理后进行业务的展示和告警业务的触发。更进一步，LinkedIn 通过自家的实时流处理系统 Samza 对这些运维数据进行实时的处理分析，生成实时的调用图分析。
- **传统的消息队列**：LinkedIn 内大量的应用系统把 Kafka 作为一个分布式消息队列进行使用。这些应用囊括了搜索、内容相关性反馈等应用，这些应用将处理后的数据通过 Kafka 传递到 Voldemort 分布式存储系统。
- **分析**：LinkedIn 会搜集所有的数据以更好地了解用户是如何使用 LinkedIn 的产品的。哪些网页被浏览，哪些内容被点击这样的信息都会发送到每个数据中心的 Kafka。这些数据被汇总起来并通过 Kafka 发送到 Hadoop 集群进行分析和每日报表生成。
- **作为其他分布式日志系统的组件**：Kafka 也被 LinkedIn 内其他分布式系统作为核心的日志组件，比如大数据仓储解决方案 Pinot。Kafka 也被分布式数据库 Espresso 用于负责数据的复制与修改。

除了 Kafka 的直接应用，LinkedIn 还开发了一些 Kafka 组件以应对其他的一些使用场景：

- MirrorMaker：让 Kafka 集群之间能同步数据。在很多情况下 LinkedIn 需要做跨数据中心的操作，对这些操作的事件记录，原有的 Kafka 无法支持，通过 MirrorMaker，Kafka 也能支持跨数据中心的事件记录传递。
- RESTful 接口：用户能通过 RESTful 接口向 Kafka 发布消费消息，而不需要开发 Java 代码的客户端。
- **审计服务**：事件一般是在一个数据中心产生的，有时候会有场景需要在另一个数据中心对该事件进行离线分析。为此，LindedIn 会把事件消息从一个数据中心同步到另一个数据中心。在消息同步的过程中，消费消息的应用需要知道什么时候这些消息被同步完，之后应用才可以开始离线处理。审计服务保证了这一点。

1.3 Kafka 的主要设计目标

Kafka 作为一种分布式的、基于发布 / 订阅的消息系统，其主要设计目标如下：

- 以时间复杂度为 $O(1)$ 的方式提供消息持久化能力，即使对 TB 级以上的数据也能保证常数时间的访问性能。
- 高吞吐率，即使在非常廉价的商用机器上也能做到单机支持每秒 100K 条消息的传输。
- 支持 Kafka Server 间的消息分区，及分布式消费，同时保证每个分区内的消息顺序传输。
- 支持离线数据处理和实时数据处理。
- 支持在线水平扩展。

1.4 为什么使用消息系统

回过头来看一下，我们为什么需要使用消息系统呢？其大致目的如下：

- **解耦**：在项目启动之初来预测将来项目会碰到什么需求，是极其困难的。消息系统在处理过程中间插入了一个隐含的、基于数据的接口层，两边的处理过程都要实现这一接口。这使得开发人员可以独立地扩展或修改两边的处理过程，只要确保它们遵守同样的接口约束即可。
- **冗余**：有些情况下，处理数据的过程会失败。除非数据被持久化，否则将造成丢失。消息队列把数据进行持久化直到数据完全处理完，通过这一方式规避了数据丢失的风险。许多消息队列所采用的“插入 - 获取 - 删除”范式中，在把一个消息从队列中删除之前，需要处理系统明确指出该消息已经被处理完毕，从而确保数据被安全保存直到使用完毕。

- **扩展性**：因为消息队列解耦了处理过程，所以增大消息入库和处理的频率是很容易的，只要另外增加处理过程即可。不需要改变代码，不需要调节参数，扩展就像调大电力按钮一样简单。
- **灵活性和峰值处理能力**：在访问量剧增的情况下，应用仍然需要继续发挥作用，但是这样的突发流量并不常见，并且以能处理这类峰值访问为标准来投入资源随时待命无疑是巨大的浪费。使用消息队列能够使关键组件顶住突发的访问压力，不会因为突发的超负荷的请求而完全崩溃。
- **可恢复性**：系统的一部分组件失效时，不会影响整个系统。消息队列降低了进程间的耦合度，所以即使一个处理消息的进程挂掉，加入队列中的消息仍然可以在系统恢复后被处理。
- **顺序保证**：在大多使用场景下，数据处理的顺序都很重要。大部分消息队列本来就是排序的，并且能保证数据会按照特定的顺序来处理。Kafka 可保证一个分区内的消息是有序的。
- **缓冲**：在任何重要的系统中，都会需要不同的处理时间的元素。例如，加载一张图片比应用过滤器花费更少的时间。消息队列通过一个缓冲层来帮助任务最高效率地执行，写入队列的处理会尽量的快速。该缓冲有助于控制和优化数据流经过系统的速度。
- **异步通信**：很多时候，用户不想也不需要立即处理消息。消息队列提供了异步处理机制，允许用户把一个消息放入队列，但并不立即处理。想向队列中放入多少消息就放多少，然后在需要的时候再去处理。

1.5　本章小结

本章先讲述了 Kafka 诞生的背景及在 LinkedIn 公司中的应用，接着讲述了 LinkedIn 设计 Kafka 的主要目标，本质上最重要的就是两点：高吞吐量和可水平扩展。最后讲述了为什么需要使用消息系统，也就是应用层使用消息系统可以解决的问题，并且这也是本书的读者需要不断思考的问题。希望通过以上四个方面的介绍使得读者能够对 Kafka 形成初步的认识，同时对自己的系统进行思考。

Chapter 2 第 2 章

Kafka 的架构

从本章起将详细探讨 Kafka 内部的实现原理：其中包括 Kafka 的基本组成、Kafka 的拓扑结构以及 Kafka 内部的通信协议。Kafka 内部的通信协议是建立在 Kafka 的拓扑结构之上的，而 Kafka 的拓扑结构是由 Kafka 的基本模块所组成的，因此本章是其他剩余章节的基础，是理解 Kafka 的关键。

2.1 Kafka 的基本组成

在 Kafka 集群中生产者将消息发送给以 Topic 命名的消息队列 Queue 中，消费者订阅发往以某个 Topic 命名的消息队列 Queue 中的消息。其中 Kafka 集群由若干个 Broker 组成，Topic 由若干个 Partition 组成，每个 Partition 里面的消息通过 Offset 来获取。

- Broker：一台 Kafka 服务器就是一个 Broker，一个集群由多个 Broker 组成，一个 Broker 可以容纳多个 Topic，Broker 和 Broker 之间没有 Master 和 Standby 的概念，它们之间的地位基本是平等的。
- Topic：每条发送到 Kafka 集群的消息都属于某个主题，这个主题就称为 Topic。物理上不同 Topic 的消息分开存储，逻辑上一个 Topic 的消息虽然保存在一个或多个 Broker 上，但是用户只需指定消息的主题 Topic 即可生产或消费数据而不需要去关心数据存放在何处。
- Partition：为了实现可扩展性，一个非常大的 Topic 可以被分为多个 Partition，从而分布到多台 Broker 上。Partition 中的每条消息都会被分配一个自增 Id（Offset）。Kafka 只保证按一个 Partition 中的顺序将消息发送给消费者，但是不保证单个 Topic 中的

多个 Partition 之间的顺序。

- Offset：消息在 Topic 的 Partition 中的位置，同一个 Partition 中的消息随着消息的写入，其对应的 Offset 也自增，其内部实现原理如图 2-1 所示。

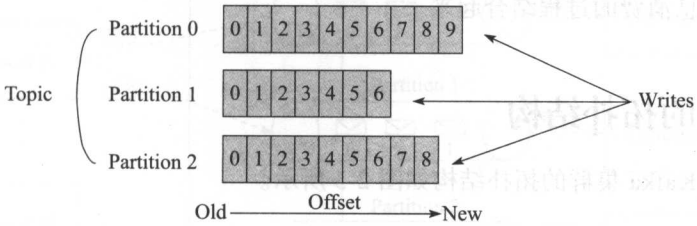


图 2-1　Topic、Partition 和 Offset 解析

- Replica：副本。Topic 的 Partition 含有 N 个 Replica，N 为副本因子。其中一个 Replica 为 Leader，其他都为 Follower，Leader 处理 Partition 的所有读写请求，与此同时，Follower 会定期地去同步 Leader 上的数据。
- Message：消息，是通信的基本单位。每个 Producer 可以向一个 Topic（主题）发布一些消息。
- Producer：消息生产者，即将消息发布到指定的 Topic 中，同时 Producer 也能决定此消息所属的 Partition：比如基于 Round-Robin（轮询）方式或者 Hash（哈希）方式等一些算法。
- Consumer：消息消费者，即向指定的 Topic 获取消息，根据指定 Topic 的分区索引及其对应分区上的消息偏移量来获取消息。
- Consumer Group：消费者组，每个 Consumer 属于一个 Consumer Group；反过来，每个 Consumer Group 中可以包含多个 Consumer。如果所有的 Consumer 都具有相同的 Consumer Group，那么消息将会在 Consumer 之间进行负载均衡。也就是说一个 Partition 中的消息只会被相同 Consumer Group 中的某个 Consumer 消费，每个 Consumer Group 消息消费是相互独立的。如果所有的 Consumer 都具有不同的 Consumer Group，则消息将会被广播给所有的 Consumer。Producer、Consumer 和 Consumer Group 之间的关系如图 2-2 所示。

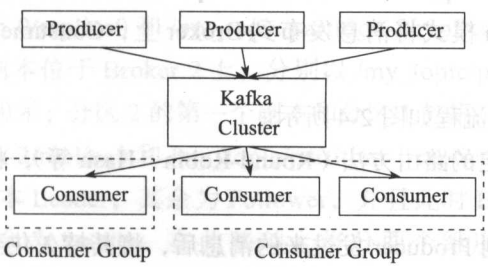


图 2-2　Producer、Consumer 和 Consumer Group 三者的关系

- Zookeeper：存放 Kafka 集群相关元数据的组件。在 Zookeeper 集群中会保存 Topic 的状态信息，例如分区的个数、分区的组成、分区的分布情况等；保存 Broker 的状态信息；保存消费者的消费信息等。通过这些信息，Kafka 很好地将消息生产、消息存储、消息消费的过程结合起来。

2.2 Kafka 的拓扑结构

一个典型的 Kafka 集群的拓扑结构如图 2-3 所示。

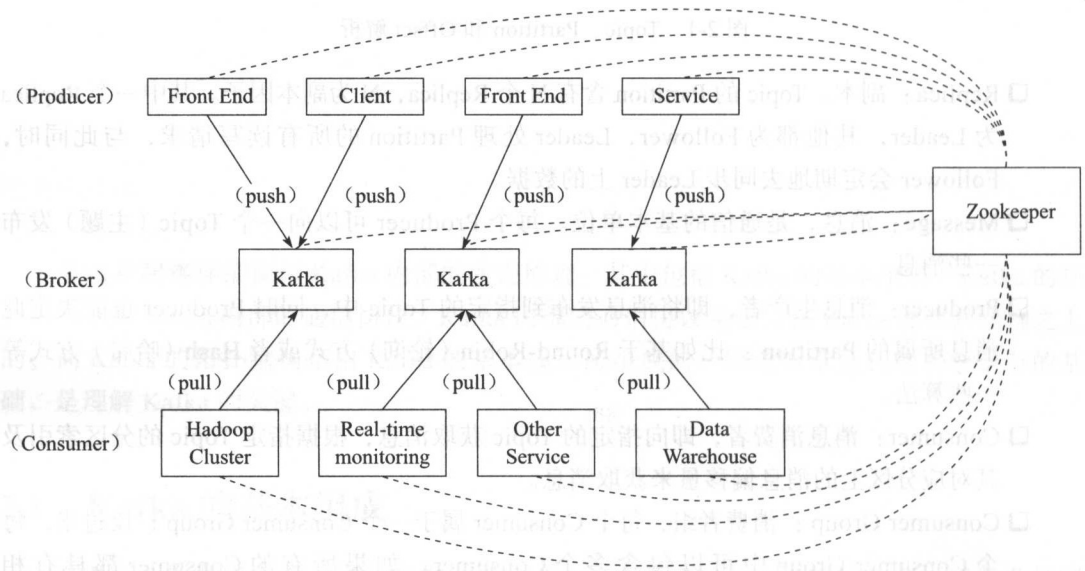


图 2-3 Kafka 拓扑结构

一个典型的 Kafka 集群中包含若干个 Producer（可以是某个模块下发的 Command，或者是 Web 前端产生的 Page View，或者是服务器日志，系统 CPU、Memory 等），若干个 Broker（Kafka 集群支持水平扩展，一般 Broker 数量越多，整个 Kafka 集群的吞吐率也就越高），若干个 Consumer Group，以及一个 Zookeeper 集群。Kafka 通过 Zookeeper 管理集群配置。Producer 使用 Push 模式将消息发布到 Broker 上，Consumer 使用 Pull 模式从 Broker 上订阅并消费消息。

一个简单的消息发送流程如图 2-4 所示。

1）Producer 根据指定的路由方法（Round-Robin、Hash 等），将消息 Push 到 Topic 的某个 Partition 里面。

2）Kafka 集群接收到 Producer 发过来的消息后，将其持久化到硬盘，并保留消息指定时长（可配置），而不关注消息是否被消费。

3）Consumer 从 Kafka 集群 Pull 数据，并控制获取消息的 Offset。

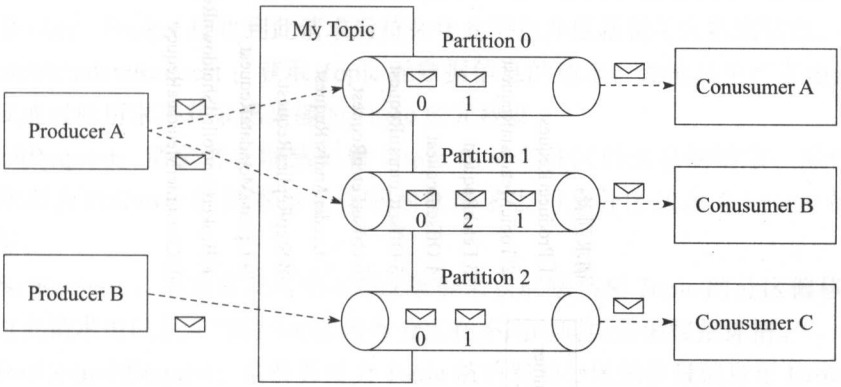


图 2-4　消息发送的简易流程

2.3　Kafka 内部的通信协议

Kafka 内部各个 Broker 之间的角色并不是完全相等的，Broker 内部负责管理分区和副本状态以及异常情况下分区的重新分配等这些功能的模块称为 KafkaController。每个 Kafka 集群中有且只有 1 个 Leader 状态的 KafkaController，当 Leader 状态的 KafkaController 出现异常时，其余的 Standby 状态下的 KafkaController 会通过 Zookeeper 选举出又一个 Leader 状态的 KafkaController，因此 Broker 又可根据 KafkaController 模块的状态进行进一步的细分。

在一个正常运行的 Kafka 集群中，生产者和 Broker 之间，消费者和 Broker 之间，Broker 和 Broker 之间不断地在进行不同网络协议的交互，正是由于各个组件不断地进行网络交互才维持了整个集群稳定的运行。其主要的网络通信协议如图 2-5 所示。

利用命令 bin/kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 2 --partitions 3 --topic my_topic 在图 2-5 中创建了以 my_topic 命名的消息主题，其分区个数为 3 个，每个分区的副本因子为 2，其中分区 0 的第一个副本和分区 2 的第二个副本位于 Broker 1 上，分别以 /my_topic/partition-0/replica0 和 /my_topic/partition-2/replica1 表示；分区 1 的第一个副本和分区 0 的第二个副本位于 Broker 2 上，分别以 /my_topic/partition-1/replica0 和 /my_topic/partition-0/replica1 表示；分区 2 的第一个副本和分区 1 的第二个副本位于 Broker 3 上，分别以 /my_topic/partition-2/replica0 和 /my_topic/partition-1/replica1 表示。假设每个分区的 replica0 为对应分区的副本 Leader，其余为 Follower，并且此时 Broker1 的 KafkaController 模块状态为 Leader，其余为 Standy。在此基础上将分两个维度来阐述图 2-5 所表达的意思：

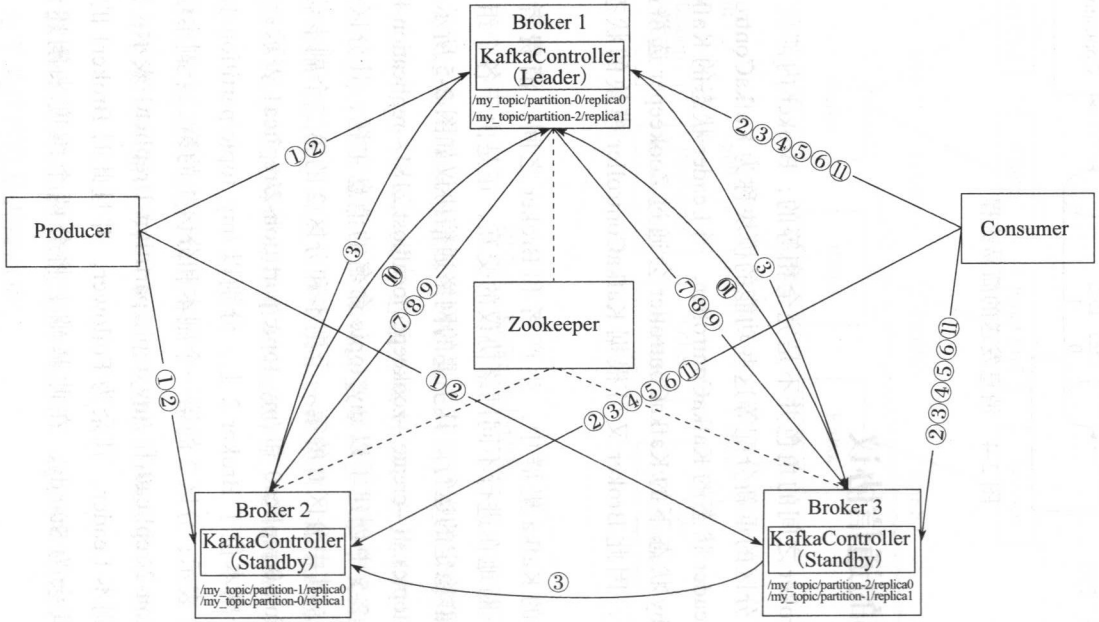


图 2-5　Kafka 内部的通信协议

请求列表：

①ProducerRequest
②TopicMetadataRequest
③FetchRequest
④OffsetRequest
⑤OffsetCommitRequest
⑥OffsetFetchRequest
⑦LeaderAndIsrRequest
⑧StopReplicaRequest
⑨UpdateMetadataRequest
⑩BrokerControlledShutdownRequest
⑪ConsumerMetadataRequest

维度一：通信协议详情

- ProducerRequest：生产者发送消息的请求，生产者将消息发送至 Kafka 集群中的某个 Broker，Broker 接收到此请求后持久化此消息并更新相关元数据信息。
- TopicMetadataRequest：获取 Topic 元数据信息的请求，无论是生产者还是消费者都需要通过此请求来获取感兴趣的 Topic 的元数据。
- FetchRequest：消费者获取感兴趣 Topic 的某个分区的消息的请求，除此之外，分区状态为 Follower 的副本也需要利用此请求去同步分区状态为 Leader 的对应副本数据。
- OffsetRequest：消费者发送至 Kafka 集群来获取感兴趣 Topic 的分区偏移量的请求，通过此请求可以获知当前 Topic 所有分区在不同时间段的偏移量详情。
- OffsetCommitRequest：消费者提交 Topic 被消费的分区偏移量信息至 Broker，Broker 接收到此请求后持久化相关偏移量信息。
- OffsetFetchRequest：消费者发送获取提交至 Kafka 集群的相关 Topic 被消费的详细信息，和 OffsetCommitRequest 相互对应。
- LeaderAndIsrRequest：当 Topic 的某个分区状态发生变化时，处于 Leader 状态的 KafkaController 发送此请求至相关的 Broker，通知其做出相应的处理。
- StopReplicaRequest：当 Topic 的某个分区被删除或者下线的时候，处于 Leader 状态的 KafkaController 发送此请求至相关的 Broker，通知其做出相应的处理。
- UpdateMetadataRequest：当 Topic 的元数据信息发生变化时，处于 Leader 状态的 KafkaController 发送此请求至相关的 Broker，通知其做出相应的处理。
- BrokerControlledShutdownRequest：当 Broker 正常下线时，发生此请求至处于 Leader 状态的 KafkaController。
- ConsumerMetadataRequest：获取保存特定 Consumer Group 消费详情的分区信息。

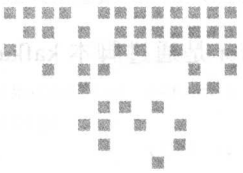
维度二：通信协议交互

- Producer 和 Kafka 集群：Producer 需要利用 ProducerRequest 和 TopicMetadataRequest 来完成 Topic 元数据的查询、消息的发送。
- Consumer 和 Kafka 集群：Consumer 需要利用 TopicMetadataRequest 请求、FetchRequest 请求、OffsetRequest 请求、OffsetCommitRequest 请求、OffsetFetchRequest 请求和 ConsumerMetadataRequest 请求来完成 Topic 元数据的查询、消息的订阅、历史偏移量的查询、偏移量的提交、当前偏移量的查询。
- KafkaController 状态为 Leader 的 Broker 和 KafkaController 状态为 Standby 的 Broker：KafkaController 状态为 Leader 的 Broker 需要利用 LeaderAndIsrRequest 请求、StopReplicaRequest 请求、UpdateMetadataRequest 请求来完成对 Topic 的管理；KafkaController 状态为 Standby 的 Broker 需要利用 BrokerControlledShutdownRequest 请求来通知 KafkaController 状态为 Leader 的 Broker 自己的下线动作。

- Broker 和 Broker 之间：Broker 相互之间需要利用 FetchRequest 请求来同步 Topic 分区的副本数据，这样才能使 Topic 分区各副本数据实时保持一致。

2.4 本章小结

本章主要讲解了有关 Kafka 消息系统的基本组成、拓扑结构和通信协议。Kafka 消息系统主要由若干个 Broker Server 组成，其中生产者向指定的 Topic 发送消息，消费者订阅发往某个 Topic 的消息。因此 Kafka 消息系统内部的通信协议主要区分为三种：生产者和 Broker 之间，消费者和 Broker 之间以及 Broker 和 Broker 之间。掌握 Kafka 消息系统内部的通信协议是理解 Kafka 内部实现原理的关键，Kafka 消息系统内部的各个模块都是通过其内部通信协议相互联系起来的。



第 3 章 Chapter 3

Broker 概述

回顾第 2 章 Kafka 的架构，Kafka 集群是由若干个 Broker 组成的，Broker 和 Broker 之间，Broker 和生产者之间，Broker 和消费者之间都存在不同的交互。因此本章将从 Broker 的启动脚本开始描述 Broker 的启动过程，以及基本阐述启动之后 Broker 内部存在的各个功能模块，包括 SocketServer、KafkaRequestHandlerPool、LogManager、ReplicaManager、OffsetManager、KafkaScheduler、KafkaApis、KafkaHealthcheck 和 TopicConfigManager 九大基本模块以及 KafkaController 集群控制管理模块。

3.1 Broker 的启动

Kafka 的安装包目录结构如图 3-1 所示。

```
drwxr-xr-x 3 hadoop hadoop  4096 Jan 29  2015 bin
drwxr-xr-x 2 hadoop hadoop  4096 Nov 16 21:38 config
drwxr-xr-x 2 hadoop hadoop  4096 Jan 29  2015 libs
-rw-r--r-- 1 hadoop hadoop 11358 Jan 29  2015 LICENSE
drwxrwxr-x 2 hadoop hadoop  4096 Mar 21 12:27 logs
-rw-r--r-- 1 hadoop hadoop   162 Jan 29  2015 NOTICE
```

图 3-1 Kafka 安装包的目录结构

bin 目录存放的是 Kafka 提供的管理工具，其中包括 Broker 的启动脚本；config 目录存放的是 Broker 的配置文件；libs 目录存放的是相关的 jar 包。

进入 bin 目录，执行以下命令后台启动 Broker：

```
nohup ./bin/kafka-server-start.sh config/server.properties &
```

可见 Broker 是通过脚本 kafka-server-start.sh 调用起来的，继续查看这个脚本的内容，如下所示：

```
if [ $# -lt 1 ];
then
  echo "USAGE: $0 [-daemon] server.properties"
  exit 1
fi
base_dir=$(dirname $0)
//省略中间步骤
exec $base_dir/kafka-run-class.sh $EXTRA_ARGS kafka.Kafka $@
```

最终执行的是 kafka.Kafka 这个类，即内部 package kafka 里面的 Kafka 类。查看源码中关于这个类的详情，如下所示：

```
object Kafka extends Logging {
  def main(args: Array[String]): Unit = {
    if (args.length != 1) {
      println("USAGE: java [options] %s
        server.properties".format(classOf[KafkaServer].getSimpleName()))
      System.exit(1)
    }
    try {
      val props = Utils.loadProps(args(0))
      val serverConfig = new KafkaConfig(props)
      KafkaMetricsReporter.startReporters(serverConfig.props)
      val kafkaServerStartable = new KafkaServerStartable(serverConfig)
      Runtime.getRuntime().addShutdownHook(new Thread() {
        override def run() = {
          kafkaServerStartable.shutdown
        }
      })
      kafkaServerStartable.startup
      kafkaServerStartable.awaitShutdown
    }
    catch {
      case e: Throwable => fatal(e)
    }
    System.exit(0)
  }
}
```

程序在 kafkaServerStartable.awaitShutdown 停住，如果继续走下去，那么 Broker 就退出了。

注意 在 Scala 语言中，class 类对象中不可有静态变量和静态方法，但是提供了“伴生对象”的功能：在和类的同一个文件中定义同名的 object 对象，所有的 main 方法都必须在 object 中被调用，来提供程序的主入口，十分简单。

其中上面的kafkaServerStartable封装了KafkaServer，最终执行startup的是KafkaServer，如下代码所示：

```
class KafkaServerStartable(val serverConfig: KafkaConfig) extends Logging {
  private val server = new KafkaServer(serverConfig)
  def startup() {
    try {
      server.startup()
      AppInfo.registerInfo()
    }
    catch {
      case e: Throwable =>
        fatal("Fatal error during KafkaServerStartable startup. Prepare to shutdown", e)
        System.exit(1)
    }
  }
  ......
}
```

在这里终于见到了Broker启动过程中最关键的类KafkaServer，接下来将围绕KafkaServer讲解里面的模块组成。

3.2 Broker内部的模块组成

首先，我们来看KafkaServer这个类包含的模块：

```
class KafkaServer(val config: KafkaConfig, time: Time = SystemTime)
extends Logging with KafkaMetricsGroup {
  this.logIdent = "[Kafka Server " + config.brokerId + "], "
  private var isShuttingDown = new AtomicBoolean(false)
  private var shutdownLatch = new CountDownLatch(1)
  private var startupComplete = new AtomicBoolean(false)
  val brokerState: BrokerState = new BrokerState
  val correlationId: AtomicInteger = new AtomicInteger(0)
  var socketServer: SocketServer = null
  var requestHandlerPool: KafkaRequestHandlerPool = null
  var logManager: LogManager = null
  var offsetManager: OffsetManager = null
  var kafkaHealthcheck: KafkaHealthcheck = null
  var topicConfigManager: TopicConfigManager = null
  var replicaManager: ReplicaManager = null
  var apis: KafkaApis = null
  var kafkaController: KafkaController = null
  val kafkaScheduler = new KafkaScheduler(config.backgroundThreads)
  var zkClient: ZkClient = null
  ......
}
```

分别是 SocketServer（监听 Socket 请求）、KafkaRequestHandlerPool（请求处理资源池）、LogManager（日志管理）、ReplicaManager（分区副本管理）、OffsetManager（偏移量管理）、KafkaScheduler（后台任务调度资源池）、KafkaApis（业务逻辑实现层）、KafkaHealthcheck（提供 Broker 健康状态）、TopicConfigManager（Topic 配置信息管理）和 KafkaController（Kafka 集群控制管理）。其相互之间的关系如图 3-2 所示。

详细说明如下：

- SocketServer：首先开启 1 个 Acceptor 线程用于监听默认端口号为 9092 上的 Socket 链接，然后当有新的 Socket 链接成功建立时会将对应的 SocketChannel 以轮询的方式转发给 N 个 Processor 线程中的某一个，并由其处理接下来该 SocketChannel 上的读写请求，其中 N=num.network.threads，默认为 3。当 Processor 线程监听来自 SocketChannel 的请求时，会将请求放置在 RequestChannel 中的请求队列；当 Processor 线程监听到 SocketChannel 请求的响应时，会将响应从 RequestChannel 中的响应队列中取出来并发送给客户端。
- KafkaRequestHandlerPool：真正处理 Socket 请求的线程池，其个数默认为 8 个，由参数 num.io.threads 决定。该线程池里面的线程 KafkaRequestHandler 从 RequestChannel 的请求队列中获取 Socket 的请求，然后调用 KafkaApis 完成真正的业务逻辑，最后将响应写回至 RequestChannel 中的响应队列，并交由 SocketServer 中对应的 Processor 线程发送给客户端。
- LogManager：Kafka 的日志管理模块。主要提供删除任何过期数据和冗余数据，刷新脏数据，对日志文件进行 Checkpoint 以及日志合并的功能。
- ReplicaManager：Kafka 的副本管理模块。主要提供针对 Topic 分区副本数据的管理功能，包括有关副本的 Leader 和 ISR 的状态变化、副本的删除、副本的监测等。其中 ISR 全称为 In-Sync Replicas，即处于同步状态的副本，在后面的章节还会详细介绍。
- OffsetManager：Kafka 的偏移量管理模块。主要提供针对偏移量的保存和读取的功能，Kafka 管理 Topic 的偏移量存在两种方式：一种为 Zookeeper，就是把偏移量提交至 Zookeeper 上；另一种为 Kafka，就是把偏移量提交至 Kafka 内部 Topic 为“__consumer_offsets”的日志里面，主要由 offsets.storage 参数决定，默认为 zookeeper。
- KafkaScheduler：Kafka 的后台任务调度资源池。提供后台定期任务的调度，主要为 LogManager、ReplicaManager 和 OffsetManager 提供调度服务。
- KafkaApis：Kafka 的业务逻辑实现层，根据不同的 Request 执行不同的操作，其中利用 LogManager、OffsetManager 和 ReplicaManager 来完成内部的处理。KafkaApis 处理的请求包括 ProducerRequest、TopicMetadataRequest、FetchRequest、OffsetRequest、OffsetCommitRequest、OffsetFetchRequest、LeaderAndIsrRequest、StopReplicaRequest、UpdateMetadataRequest、BrokerControlledShutdownRequest 和 ConsumerMetadataRequest。

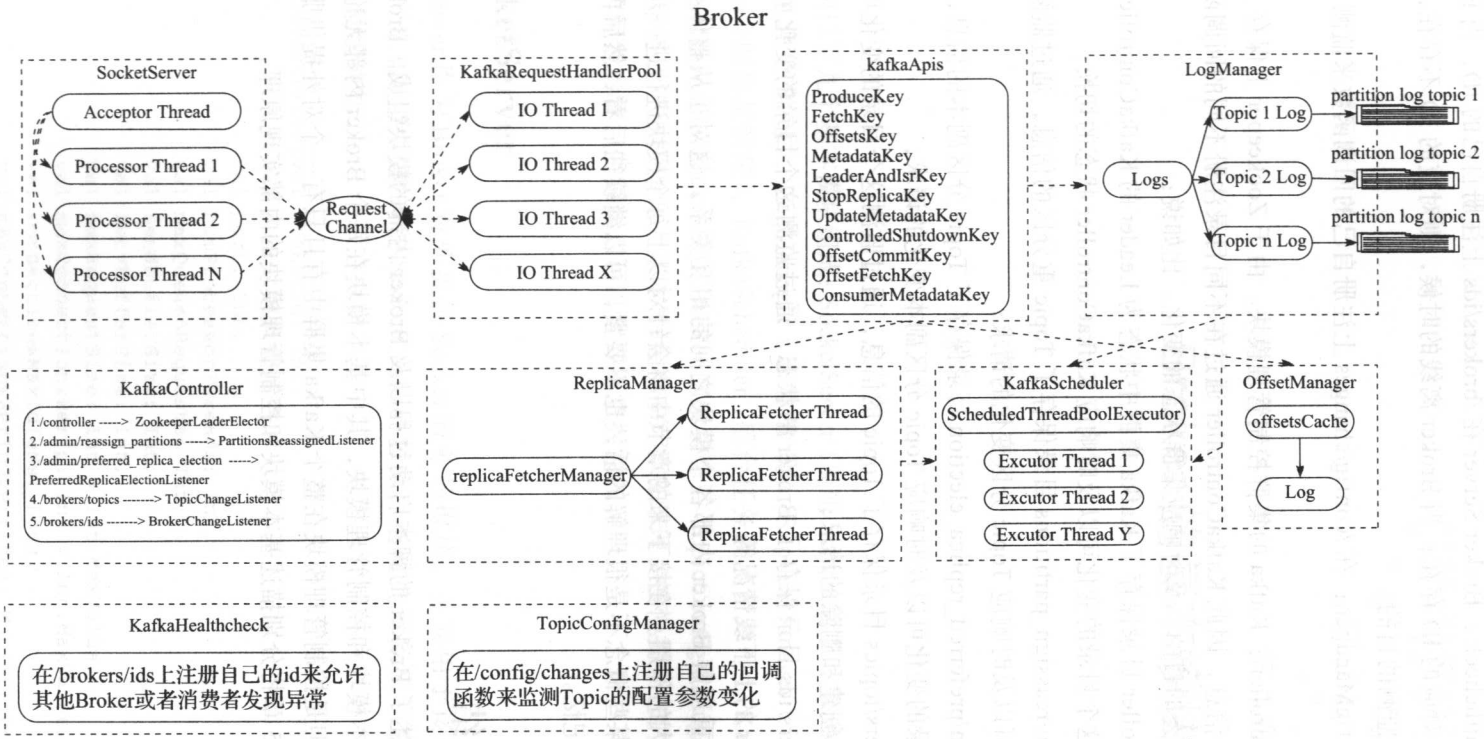


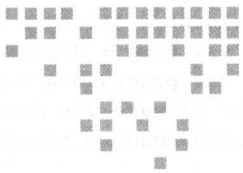
图 3-2　Broker 内部模块之间的关系

- KafkaHealthcheck：Broker Server 在 /brokers/ids 上注册自己的 ID，当 Broker 在线的时候，则对应的 ID 存在；当 Broker 离线的时候，则对应的 ID 不存在，以此来达到集群状态监测的目的。
- TopicConfigManager：在 /config/changes 上注册自己的回调函数来监测 Topic 配置信息的变化。
- KafkaController：Kafka 的集群控制管理模块。由于 Zookeeper 上保存了 Kafka 集群的元数据信息，因此 KafkaController 通过在不同目录注册不同的回调函数来达到监测集群状态的目的，及时响应集群状态的变化。比如说：
  1）/controller 目录保存了 Kafka 集群中状态为 Leader 的 KafkaController 标识，通过监测这个目录的变化可以及时响应 KafkaController 状态的切换；
  2）/admin/reassign_partitions 目录保存了 Topic 重分区的信息，通过监测这个目录的变化可以及时响应 Topic 分区变化的请求；
  3）/admin/preferred_replica_election 目录保存了 Topic 分区副本的信息，通过监测这个目录的变化可以及时响应 Topic 分区副本变化的请求；
  4）/brokers/topics 目录保存了 Topic 的信息，通过监测这个目录的变化可以及时响应 Topic 创建和删除的请求；
  5）/brokers/ids 目录保存了 Broker 的状态，通过监测这个目录的变化可以及时响应 Broker 的上下线情况等。

希望读者着重理解 Broker 内部各个模块之间的相互关系，这对于从整体上把握 Kafka 架构会起到比较大的作用。在接下来的章节中将会针对以上每个模块进行进一步的讲解，如果在这一章节对某些概念不是很理解的话，也不要紧，可以继续往后看，然后再回过头看整体图，以便加深记忆。

3.3 本章小结

本章大致讲述了 Broker 的脚本启动过程以及 Broker 内部的模块组成。Broker 的模块组成主要区分为基本模块和控制管理模块，其中基本模块在每个 Broker 内部无论对内还是对外都提供服务，但是控制管理模块在整个 Kafka 集群中有且只有一个对外提供服务。在接下去的第 4 章和第 5 章将分别描述基本模块和控制管理模块的内部实现原理。



第 4 章 Chapter 4

Broker 的基本模块

Broker 由以下九个基本模块组成：SocketServer（监听 Socket 请求）、KafkaRequestHandlerPool（请求处理资源池）、LogManager（日志管理）、ReplicaManager（分区副本管理）、OffsetManager（偏移量管理）、KafkaScheduler（后台任务调度资源池）、KafkaApis（业务逻辑实现层）、KafkaHealthcheck（提供 Broker 健康状态）、TopicConfigManager（Topic 配置信息管理）。本章将层层剖析这九大基本模块内部的实现原理，慢慢揭开它们的面纱，并且逐渐暴露内部的配置参数，使得读者可以更好地掌握它们。

4.1 SocketServer

SocketServer 作为 Broker 对外提供 Socket 服务的模块，主要用于接收 Socket 连接的请求，然后产生相应为之服务的 SocketChannel 对象，通过此对象来和客户端相互通信。SocketServer 的组成如下：

```
class SocketServer(val brokerId: Int,
                   val host: String,
                   val port: Int,
                   val numProcessorThreads: Int,
                   val maxQueuedRequests: Int,
                   val sendBufferSize: Int,
                   val recvBufferSize: Int,
                   val maxRequestSize: Int = Int.MaxValue,
                   val maxConnectionsPerIp: Int = Int.MaxValue,
                   val connectionsMaxIdleMs: Long,
                   val maxConnectionsPerIpOverrides: Map[String, Int] )
```

```
  extends Logging with KafkaMetricsGroup {
  this.logIdent = "[Socket Server on Broker " + brokerId + "], "
  private val time = SystemTime
  private val processors = new Array[Processor](numProcessorThreads)
  @volatile private var acceptor: Acceptor = null
  val requestChannel = new RequestChannel(numProcessorThreads, maxQueuedRequests)
  ......
}
```

它内部主要包括三个模块：1）Acceptor 主要用于监听 Socket 的连接；2）Processor 主要用于转发 Socket 的请求和响应；3）RequestChannel 主要用于缓存 Socket 的请求和响应。

Acceptor 的初始化过程如下：

```
private[kafka] class Acceptor(val host: String,
                              val port: Int, private val processors: Array[Processor],
                              val sendBufferSize: Int,
                              val recvBufferSize: Int,
                              connectionQuotas: ConnectionQuotas)
  extends AbstractServerThread(connectionQuotas) {
  // 开启 Socket 服务
  val serverChannel = openServerSocket(host, port)
  def run() {
    // 注册 Accept 事件
    serverChannel.register(selector, SelectionKey.OP_ACCEPT);
    startupComplete()
    var currentProcessor = 0
    // 监听 Accept 事件
    while(isRunning) {
      val ready = selector.select(500)
      if(ready > 0) {
        val keys = selector.selectedKeys()
        val iter = keys.iterator()
        while(iter.hasNext && isRunning) {
          var key: SelectionKey = null
          try {
            key = iter.next
            iter.remove()
            if(key.isAcceptable)
              accept(key, processors(currentProcessor))
            else
              throw new IllegalStateException("Unrecognized key state for acceptor
                thread.")
            currentProcessor = (currentProcessor + 1) % processors.length
          } catch {
            case e: Throwable => error("Error while accepting connection", e)
          }
        }
      }
    }
    debug("Closing server socket and selector.")
```

```
      swallowError(serverChannel.close())
      swallowError(selector.close())
      shutdownComplete()
    }
    ......
  }
```

其主要步骤如下：

1）开启 Socket 服务。

2）注册 Accept 事件。

3）监听此 ServerChannel 上的 ACCEPT 事件，当其发生时，将其以轮询（Round Robin）的方式把对应的 SocketChannel 转交给 Processor 处理线程。

注意　OP_ACCEPT 为 NIO 中的事件，当此事件发生时，表示服务器监听到了客户连接，服务器可以接收这个连接了。

Processor 的初始化过程大致如下：

```
private[kafka] class Processor(val id: Int,
                               val time: Time,
                               val maxRequestSize: Int,
                               val aggregateIdleMeter: Meter,
                               val idleMeter: Meter,
                               val totalProcessorThreads: Int,
                               val requestChannel: RequestChannel,
                               connectionQuotas: ConnectionQuotas,
                               val connectionsMaxIdleMs: Long)
  extends AbstractServerThread(connectionQuotas) {
  private val newConnections = new ConcurrentLinkedQueue[SocketChannel]()
  override def run() {
    startupComplete()
    while(isRunning) {
      // 针对新的连接，注册其上的 OP_READ 事件
      configureNewConnections()
      // 从 RequestChannel 获取响应产生 OP_WRITE 事件
      processNewResponses()
      val startSelectTime = SystemTime.nanoseconds
      val ready = selector.select(300)
      currentTimeNanos = SystemTime.nanoseconds
      val idleTime = currentTimeNanos - startSelectTime
      idleMeter.mark(idleTime)
      aggregateIdleMeter.mark(idleTime / totalProcessorThreads)
      trace("Processor id " + id + " selection time = " + idleTime + " ns")
      // 监听 selector 上的 OP_READ 事件和 OP_WRITE 事件
      if(ready > 0) {
        val keys = selector.selectedKeys()
```

```
        val iter = keys.iterator()
        while(iter.hasNext && isRunning) {
          var key: SelectionKey = null
          try {
            key = iter.next
            iter.remove()
            if(key.isReadable)
              read(key)
            else if(key.isWritable)
              write(key)
            else if(!key.isValid)
              close(key)
            else
              throw new IllegalStateException("Unrecognized key state for processor
                thread.")
          } catch {
            case e: EOFException => {
              info("Closing socket connection to %s.".format(channelFor(key).
                socket.getInetAddress))
              close(key)
            }
            case e: InvalidRequestException => {
              info("Closing socket connection to %s due to invalid request:
                %s".format(channelFor(key).socket.getInetAddress, e.getMessage))
              close(key)
            }
            case e: Throwable => {
              error("Closing socket for " + channelFor(key).socket.getInetAddress +
                " because of error", e)
              close(key)
            }
          }
        }
      }
      maybeCloseOldestConnection
    }
    debug("Closing selector.")
    closeAll()
    swallowError(selector.close())
    shutdownComplete()
  }
```

其中 newConnections 保存了由 Acceptor 线程转移过来的 SocketChannel 对象，主要步骤如下：

1）当有新的 SocketChannel 对象进来的时候，注册其上的 OP_READ 事件以便接收客户端的请求。

2）从 RequestChannel 中的响应队列获取对应客户端请求的响应，然后产生 OP_WRITE 事件。

3）监听 selector 上的事件。如果是读事件，说明有新的 request 到来，需要转移给 RequestChannel 的请求队列；如果是写事件，说明之前的 request 已经处理完毕，需要从 RequestChannel 的响应队列获取响应并发送回客户端；如果是关闭事件，说明客户端已经关闭了该 Socket 连接，此时服务端也应该释放相应资源。

注意　OP_WRITE 为 NIO 中的写事件，如果 SelectKey 注册了写事件，不在合适的时间去除掉，会一直触发写事件，因为写事件是代码触发的，其他详情可以翻阅 NIO 相关资料。

除此之外，SocketServer 为了防止空闲连接大量存在，采用了 LRU（Least Recently Used）算法，即最近最少使用算法，会将长时间没有交互的 SocketChannel 对象关闭，及时释放资源。因此 Processor 仅仅是起到了接收 Request，发送 Response 的作用，其处理 Request 的具体业务逻辑是由 KafkaApis 层负责的，并且两者之间是通过 RequestChannel 相互联系起来的。

RequestChannel 本质上就是为了解耦 SocketServer 和 KafkaApis 两个模块，内部包含 Request 的阻塞队列和 Response 的阻塞队列，其具体实现如下：

```
class RequestChannel(val numProcessors: Int, val queueSize: Int) extends
  KafkaMetricsGroup {
  private var responseListeners: List[(Int) => Unit] = Nil
  private val requestQueue = new ArrayBlockingQueue[RequestChannel.Request]
    (queueSize)
  private val responseQueues = new Array[BlockingQueue[RequestChannel.Response]]
    (numProcessors)
  for(i <- 0 until numProcessors)
    responseQueues(i) = new LinkedBlockingQueue[RequestChannel.Response]()
  def sendRequest(request: RequestChannel.Request) {
    requestQueue.put(request)
  }
  def sendResponse(response: RequestChannel.Response) {
    responseQueues(response.processor).put(response)
    for(onResponse <- responseListeners)
      onResponse(response.processor)
  }
}
```

RequestChannel 内部包含了 1 个 Request 的阻塞队列和 numProcessors 个 Response 的阻塞队列，其中 numProcessors =num.network.threads，默认为 3 个。Processor 线程通过监听 OP_READ 事件将 Request 转移到 RequestChannel 内部的 Request 阻塞队列，KafkaRequestHandlerPool 内部的 KafkaRequestHandler 线程从 RequestChannel 内部的 Request 阻塞队列取出 Request 进行处理，然后将对应的 Response 放回至 RequestChannel 内部的 Response 阻塞队列，并触发 Processor 线程监听的 OP_WRITE 事件，最后由 Processor 线程将 Response 发送至客户端，相互之间具体的关系如图 4-1 所示。

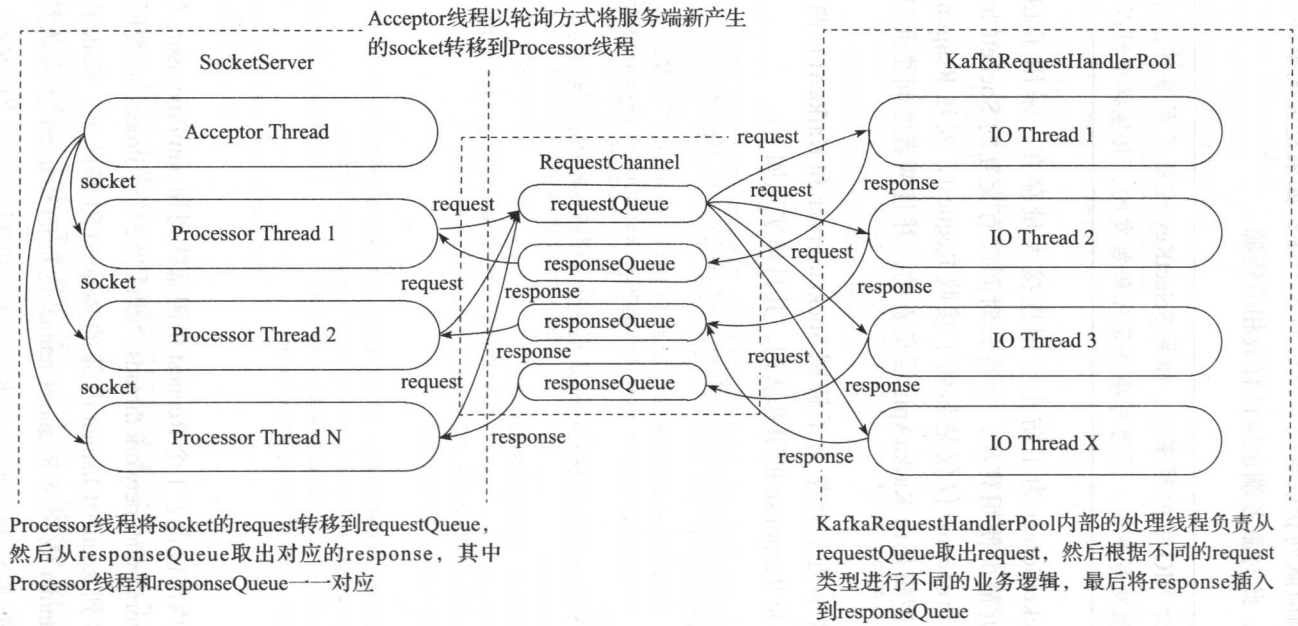


图 4-1　SocketServer 内部模块之间的相互关系

其中 Request 对象的 processor 变量标识了该 Request 来源于哪个 Processor 线程，假设 Request.processor=i，则该 Request 的 Response 最终会存放在第 i 个 Response 阻塞队列，即 responseQueue[i]。Request 对象的属性如下：

```
case class Request(processor: Int,
                   requestKey: Any,
                   private var buffer: ByteBuffer,
                   startTimeMs: Long,
                   remoteAddress: SocketAddress = new InetSocketAddress(0)) {
}
```


Scala 中的构造函数是和类的定义混杂在一起的，即定义类的时候，就指明了这个类的成员变量参数。还有，在类中，除了 def 定义的成员函数之外的所有操作，都可以看作是构造函数的行为组成部分，不管是变量赋值还是函数调用。而 Java 的类定义和构造函数的定义是分开的，因此在 Request 类里面包含变量名为 processor 的成员。

SocketServer 模块至此已经分析完毕，可见它仅仅负责以下三个方面：1）建立 Socket，保持和客户端的通信；2）转发客户端的 Request；3）返回 Response 给客户端。最后通过 RequestChannel 与其他模块解耦。

4.2 KafkaRequestHandlerPool

KafkaRequestHandlerPool 本质上就是一个线程池，里面包含了 num.io.threads 个 IO 处理线程，默认为 8 个，其内部实现如下：

```
KafkaRequestHandlerPool(val brokerId: Int,
                        val requestChannel: RequestChannel,
                        val apis: KafkaApis,
                        numThreads: Int)
  extends Logging with KafkaMetricsGroup {
  ......
  val threads = new Array[Thread](numThreads)
  val runnables = new Array[KafkaRequestHandler](numThreads)
  for(i <- 0 until numThreads) {
    runnables(i) = new KafkaRequestHandler(i,
                                           brokerId,
                                           aggregateIdleMeter,
                                           numThreads,
                                           requestChannel,
                                           apis)
    threads(i) = Utils.daemonThread("kafka-request-handler-" + i, runnables(i))
```

```
      threads(i).start()
    }
    ......
  }
```

KafkaRequestHandlerPool 在内部启动了若干个 KafkaRequestHandler 处理线程，并将 RequestChannel 对象和 KafkaApis 对象传递给了 KafkaRequestHandler 处理线程，因为 KafkaRequestHandler 需要从前者的 requestQueue 中取出 Request，并且利用后者来完成具体的业务逻辑。请看 KafkaRequestHandler 具体的实现：

```
class KafkaRequestHandler(id: Int,
                          brokerId: Int,
                          val aggregateIdleMeter: Meter,
                          val totalHandlerThreads: Int,
                          val requestChannel: RequestChannel,
                          apis: KafkaApis)
  extends Runnable with Logging {
  def run() {
    while(true) {
      try {
        var req : RequestChannel.Request = null
        while (req == null) {
          val startSelectTime = SystemTime.nanoseconds
          req = requestChannel.receiveRequest(300)
          val idleTime = SystemTime.nanoseconds - startSelectTime
          aggregateIdleMeter.mark(idleTime / totalHandlerThreads)
        }
        if(req eq RequestChannel.AllDone) {
          return
        }
        req.requestDequeueTimeMs = SystemTime.milliseconds
        apis.handle(req)
      } catch {
        case e: Throwable => error("Exception when handling request", e)
      }
    }
  }
}
```

其主要步骤如下：

1）通过调用 requestChannel.receiveRequest 从 requestChannel 的 Request 阻塞队列中获取请求，如果获取不到，则一直在 while 循环之中不断尝试，直至获取到新的请求为止。

2）判断请求类型，如果是 RequestChannel.AllDone 类型的话，则退出线程；否则调用 apis 完成请求的处理，并由 apis 负责将对应的响应放入 requestChannel 的 Response 阻塞队列。

因此 SocketServer 的 Processor 线程和 KafkaRequestHandlerPool 的 KafkaRequestHandler

线程利用 SocketServer 内部的 RequestChannel 实现了一个简单的生成者和消费者模型，即如图 4-2 所示。

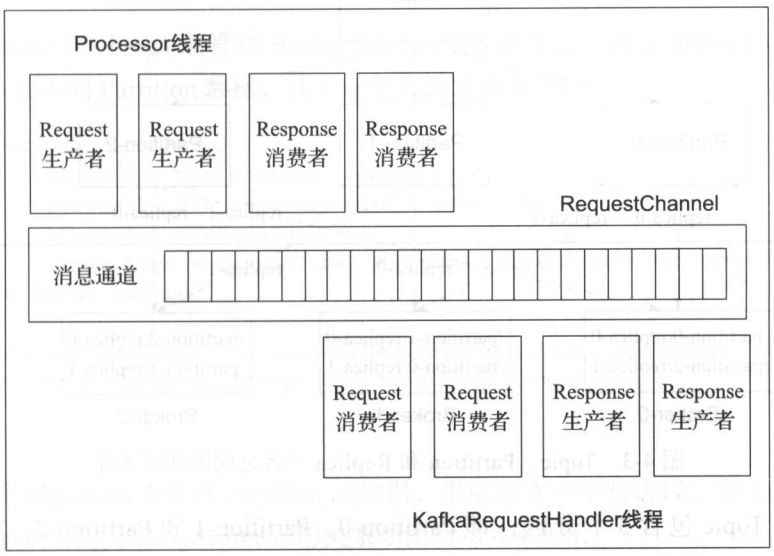


图 4-2　简单的生产者和消费者模型

4.3　KafkaApis

KafkaApis 负责具体的业务逻辑，它主要和 Producer、Consumer、Broker Server 交互，搞清楚这些交互就彻底知道了 Broker Server 的所有行为。

KafkaApis 主要依赖以下四个组件来完成具体的业务逻辑：

- LogManager 提供针对 Kafka 的 Topic 日志的读取和写入功能。
- ReplicaManager 提供针对 Topic 的分区副本数据的同步功能。
- OffsetManager 提供针对提交至 Kafka 的偏移量的管理功能。
- KafkaScheduler 为其他模块提供定时任务的调度和管理功能。

首先，我们来分析以上四个模块的具体实现，然后针对不同类型的 Request 来分析 KafkaApis 内部的具体实现逻辑。

4.3.1　LogManager

LogManager 负责提供 Broker Server 上 Topic 的分区数据读取和写入功能，负责读取和写入位于 Broker Server 上的所有分区副本数据；如果 Partition 有多个 Replica，则每个 Broker Server 不会存在相同 Partition 的 Replica；如果存在的话，一旦遇到 Broker Server 下线，则会立刻丢失 Partition 的多份副本，失去了一定的可靠性。Topic、Partition 和 Replica

三者之间的相互关系如图 4-3 所示。

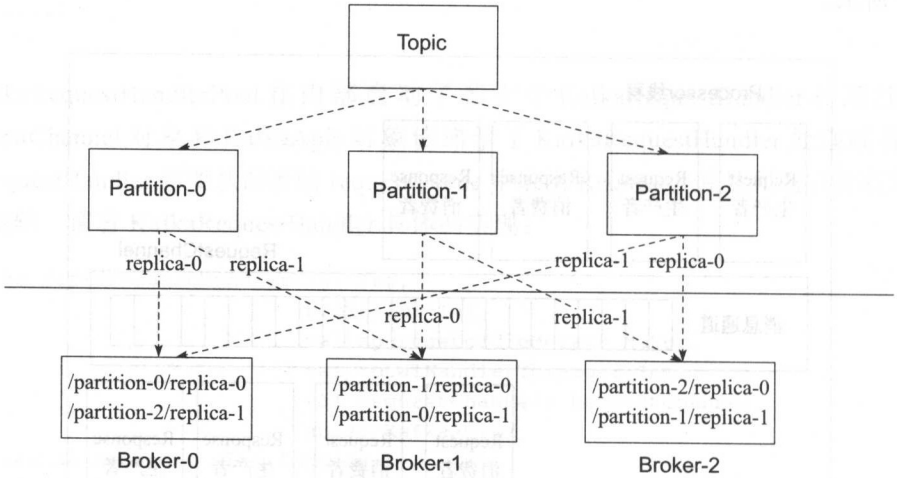


图 4-3 Topic、Partition 和 Replica 三者之间的相互关系

图 4-3 中 Topic 包含 3 个分区，即 Partition-0、Partition-1 和 Partition-2。每个 Partition 包含 2 个 Replica，因此 Partition-0 的两个 Replica 分布在 Broker-0 和 Broker-1 上，Partition-1 的两个 Replica 分布在 Broker-1 和 Broker-2 上，Partition-2 的两个 Replica 分布在 Broker-2 和 Broker-0 上。Broker Server 内部的 LogManager 管理的就是属于该 Broker 上的 Topic 的 Partition 数据，也就是说 Broker-0 内部的 LogManager 负责该 Broker 上的 Partition-0 和 Partition-2 的数据，Broker-1 内部的 LogManager 负责该 Broker 上的 Partition-0 和 Partition-1 的数据，Broker-2 内部的 LogManager 负责该 Broker 上的 Partition-2 和 Partition-1 的数据。

4.3.1.1 Kafka 的日志组成

Kafka 的日志指的就是 Topic 对应的 Partition 数据。因此我们首先来看一下 LogManager 的组成。LogManager 内部的组成大致如下：

```
LogManager(val logDirs: Array[File],
           val topicConfigs: Map[String, LogConfig],
           val defaultConfig: LogConfig,
           val cleanerConfig: CleanerConfig,
           ioThreads: Int,
           val flushCheckMs: Long,
           val flushCheckpointMs: Long,
           val retentionCheckMs: Long,
           scheduler: Scheduler,
           val brokerState: BrokerState,
           private val time: Time) extends Logging {
  val RecoveryPointCheckpointFile = "recovery-point-offset-checkpoint"
  val LockFile = ".lock"
  val InitialTaskDelayMs = 30*1000
```

```
  private val logCreationOrDeletionLock = new Object
  private val logs = new Pool[TopicAndPartition, Log]()
  ......
}
```

LogManager 利用 logs 来管理 Broker Server 内部的日志，通过 TopicAndPartition 来索引不同 Topic 的不同 Partition 数据，其 Log 的大致组成如下：

```
class Log(val dir: File,
          @volatile var config: LogConfig,
          @volatile var recoveryPoint: Long = 0L,
          scheduler: Scheduler,
          time: Time = SystemTime) extends Logging with KafkaMetricsGroup {
  import kafka.log.Log._
  ......
  private val segments: ConcurrentNavigableMap[java.lang.Long, LogSegment] =
    new ConcurrentSkipListMap[java.lang.Long, LogSegment]
  ......
}
```

Log 利用 segments 来管理 Partition 的数据，里面包含多个日志段，即 LogSegment。因此 LogManager、Log 和 LogSegment 的关系如图所示 4-4 所示。

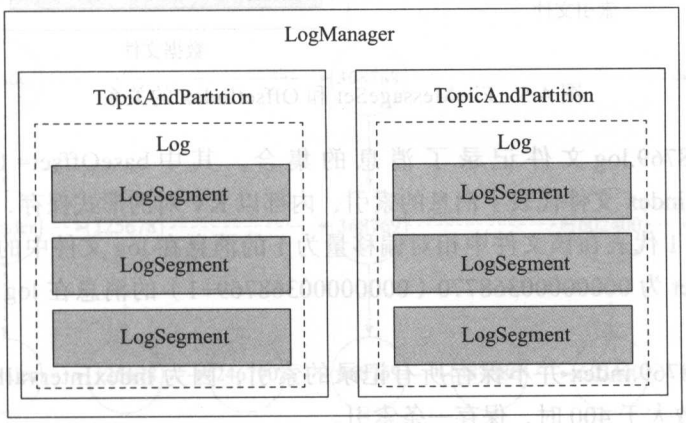


图 4-4　LogSegment，Log 和 LogSegment 的关系

那么 LogSegment 又是由什么组成的呢？且看 LogSegment 的实现，如下所示：

```
class LogSegment(val log: FileMessageSet,
                 val index: OffsetIndex,
                 val baseOffset: Long,
                 val indexIntervalBytes: Int,
                 val rollJitterMs: Long,
                 time: Time) extends Logging {
  ......
}
```

其中 log 代表的是消息集合，每条消息都有一个 Offset，这是针对 Partition 中的偏移量；index 代表的是消息的索引信息，以 KV 对的形式记录，其中 K 为消息在 log 中的相对偏移量，V 为消息在 log 中的绝对位置；baseOffset 代表的是该 LogSegment 日志段的起始偏移量；indexIntervalBytes 代表的是索引的粒度，即写入多少字节之后生成一条索引，OffsetIndex 不会保存每条消息的索引，因此其索引文件是一个稀疏索引文件，具体的表现形式如图 4-5 所示。

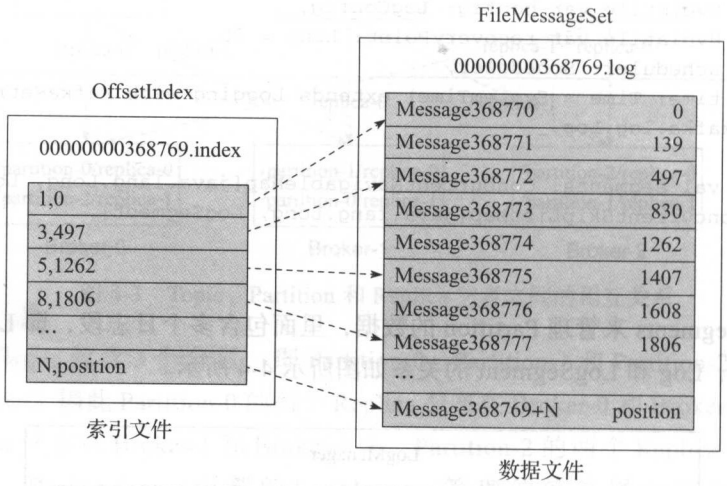


图 4-5 FileMessageSet 和 OffsetIndex 的关系

00000000368769.log 文件记录了消息的集合，其中 baseOffset= 00000000368769。00000000368769.index 文件代表了消息的索引，内部以 KV 对的形式保存，比如第一条索引记录（1,0）中的 1 代表在该文件中相对偏移量为 1 的消息在 log 文件中的物理偏移量，即绝对偏移量 Offset 为 00000000368770（00000000368769+1）的消息在 log 中的起始物理位置为 0。

00000000368769.index 并不保存所有记录的索引，因为 indexIntervalBytes=400，即当累计消息的字节数大于 400 时，保存一条索引。

4.3.1.2 Kafka 的消息读取

既然 Broker Server 上针对 Topic 的 Partition 都会保存一堆 log 文件，每个 log 文件都有其对应的 index 文件，那么 Broker Server 是如何定位具体的消息的呢？且看 Log 提供的 read 函数：

```
def read(startOffset: Long, maxLength: Int, maxOffset: Option[Long] = None):
  FetchDataInfo = {
  val next = nextOffsetMetadata.messageOffset
  // 如果是当前最新的 Offset，则无数据读取
  if(startOffset == next)
```

```
    return FetchDataInfo(nextOffsetMetadata, MessageSet.Empty)
  // 根据 startOffset 定位位于哪个 LogSegment
  var entry = segments.floorEntry(startOffset)
  /* 异常判断，如果 startOffset 大于当前最大的偏移量或没有找到具体的 LogSegment，则抛异常 */
  if(startOffset > next || entry == null)
    throw new OffsetOutOfRangeException("Request for offset %d but we only
      have log segments in the range %d to %d.".format(startOffset, segments.
      firstKey, next))
  // 调用 LogSegment 的 read 方法读取到具体的信息
  while(entry != null) {
    val fetchInfo = entry.getValue.read(startOffset, maxOffset, maxLength)
    if(fetchInfo == null) {
      entry = segments.higherEntry(entry.getKey)
    } else {
      return fetchInfo
    }
  }
  FetchDataInfo(nextOffsetMetadata, MessageSet.Empty)
}
```

我们考虑 startOffset 正常情况下的流程。主要分两个步骤：

步骤 1　根据 startOffset 定位位于哪个 LogSegment，其中 segments 的数据结构为 ConcurrentSkipListMap，是一种跳跃表的数据结构，其数据查找的方式如图 4-6 所示。

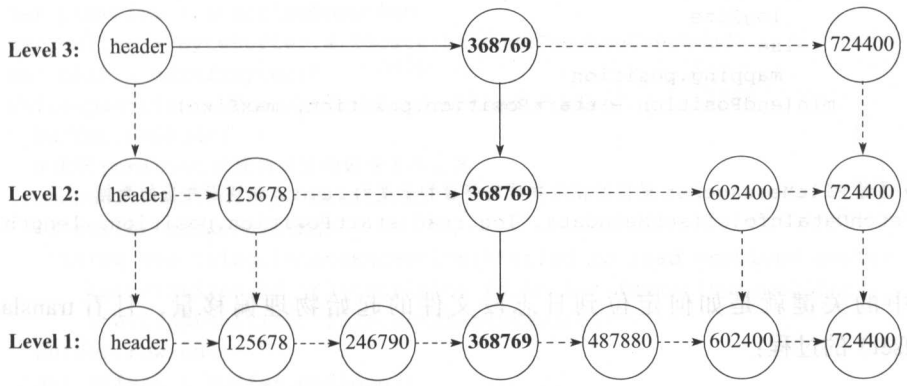


图 4-6　定位 LogSegment 的过程

跳跃表分为许多层（level），每一层都可以看作是数据的索引，这些索引的意义就是加快跳跃表查找数据的速度。每一层的数据都是有序的，上一层数据是下一层数据的子集，并且第一层（Level 1）包含了全部的数据；层次越高，跳跃性越大，包含的数据越少。跳跃表包含一个表头（header），查找数据是从上往下，从左往右进行。现在需要找出小于等于 startOffset 为 00000000368773 的节点，其查找过程如图中箭头所示。

步骤 2　当找到具体的 LogSegment 的时候，则调用其提供的 read 方法，如下所示：

```
def read(startOffset: Long, maxOffset: Option[Long], maxSize: Int): FetchDataInfo = {
```

```
    if(maxSize < 0)
      throw new IllegalArgumentException("Invalid max size for log read (%d)".
        format(maxSize))
    val logSize = log.sizeInBytes
    // 通过 startOffset 找到位于 log 中具体的物理位置，以字节为单位
    val startPosition = translateOffset(startOffset)
    if(startPosition == null)
      return null
    // 组装 offset 元数据信息
    val offsetMetadata = new LogOffsetMetadata(startOffset, this.baseOffset,
      startPosition.position)
    // 如果设置了 maxOffset，则根据其具体值计算实际需要读取的字节数
    if(maxSize == 0)
      return FetchDataInfo(offsetMetadata, MessageSet.Empty)
    val length =
      maxOffset match {
        case None =>
          maxSize
        case Some(offset) => {
          if(offset < startOffset)
            throw new IllegalArgumentException("Attempt to read with a maximum
              offset (%d) less than the start offset (%d).".format(offset, startOffset))
          val mapping = translateOffset(offset, startPosition.position)
          val endPosition =
            if(mapping == null)
              logSize
            else
              mapping.position
          min(endPosition - startPosition.position, maxSize)
        }
      }
    // 通过 FileMessageSet 提供的指定物理偏移量和长度的 read 方法读取相应的数据
    FetchDataInfo(offsetMetadata, log.read(startPosition.position, length))
  }
```

其中的关键就是如何定位到日志段文件的起始物理偏移量，且看 translateOffset（startOffset）的过程：

```
  private[log] def translateOffset(offset: Long, startingFilePosition: Int = 0):
    OffsetPosition = {
    /* 通过 index 索引信息定位到小于等于 startOffset 的最近记录位置，利用的是二分查找算法 */
    val mapping = index.lookup(offset)
    /* 从小于等于 startOffset 的最近记录位置开始往后读取数据，直到读取到偏移量为
    startOffset 的消息 */
    log.searchFor(offset, max(mapping.position, startingFilePosition))
  }
```

在这个过程中主要分两个阶段：

阶段 1 通过二分查找算法找到小于等于 startOffset 的最近索引记录位置，其过程如图 4-7 所示。

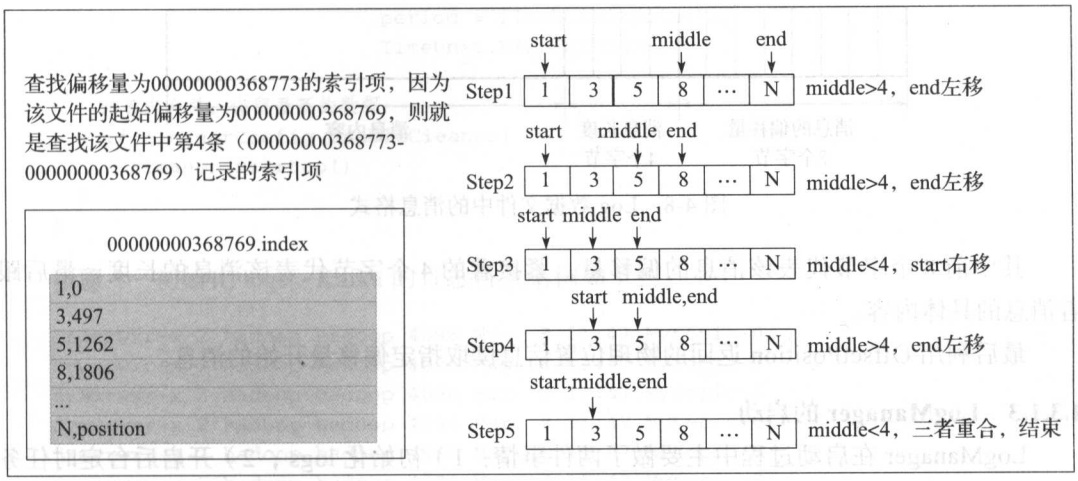


图 4-7　二分查找算法查找索引项

阶段 2　打开 log 文件，将文件指针移至偏移量为 497 的位置，然后从这个位置开始往下读取数据，直至读取到消息偏移量为 00000000368773 的位置，返回其真实的物理偏移量，如下所示：

```
def searchFor(targetOffset: Long, startingPosition: Int): OffsetPosition = {
  var position = startingPosition
  val buffer = ByteBuffer.allocate(MessageSet.LogOverhead)
  val size = sizeInBytes()
  while(position + MessageSet.LogOverhead < size) {
    buffer.rewind()
    // 读取 position 位置的消息的偏移量和长度
    channel.read(buffer, position)
    if(buffer.hasRemaining)
      throw new IllegalStateException("Failed to read complete buffer for
        targetOffset %d startPosition %d in %s".format(targetOffset,
        startingPosition, file.getAbsolutePath))
    buffer.rewind()
    val offset = buffer.getLong()
    if(offset >= targetOffset)
      return OffsetPosition(offset, position)
    // 获取消息的长度，计算下一条消息的起始位置
    val messageSize = buffer.getInt()
    if(messageSize < Message.MessageOverhead)
      throw new IllegalStateException("Invalid message size: " + messageSize)
    position += MessageSet.LogOverhead + messageSize
  }
  null
}
```

其 Log 数据文件中的消息格式如图 4-8 所示。

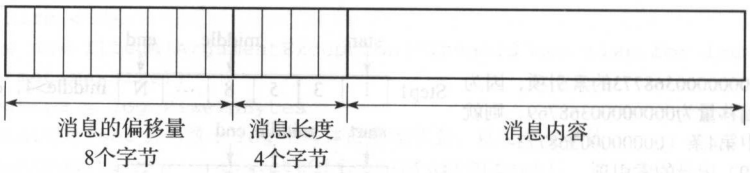


图 4-8 Log 数据文件中的消息格式

其中前 8 个字节代表该消息的偏移量，紧接着的 4 个字节代表该消息的长度，最后跟着消息的具体内容。

最后利用 OffsetPosition 返回的物理位置信息读取指定偏移量开始的消息。

4.3.1.3 LogManager 的启动

LogManager 在启动过程中主要做了两件事情：1）初始化 logs；2）开启后台定时任务和维护线程，如下所示：

```
class LogManager(val logDirs: Array[File], val topicConfigs: Map[String, LogConfig],
                 val defaultConfig: LogConfig, val cleanerConfig: CleanerConfig,
                 ioThreads: Int, val flushCheckMs: Long,
                 val flushCheckpointMs: Long, val retentionCheckMs: Long,
                 scheduler: Scheduler,val brokerState: BrokerState,
                 private val time: Time) extends Logging {
  ......
  // 初始化 logs，即 new Pool[TopicAndPartition, Log]() 对象
  private val recoveryPointCheckpoints = logDirs.map(dir => (dir, new
    OffsetCheckpoint(new File(dir, RecoveryPointCheckpointFile)))).toMap
  loadLogs()
  ......
  def startup() {
    if(scheduler != null) {
      info("Starting log cleanup with a period of %d ms.".format(retentionCheckMs))
      // 删除过期的数据和冗余的数据
      scheduler.schedule("kafka-log-retention",
                         cleanupLogs,
                         delay = InitialTaskDelayMs,
                         period = retentionCheckMs,
                         TimeUnit.MILLISECONDS)
      info("Starting log flusher with a default period of %d ms.".format(flushCheckMs))
      // flush 脏数据
      scheduler.schedule("kafka-log-flusher",
                         flushDirtyLogs,
                         delay = InitialTaskDelayMs,
                         period = flushCheckMs,
                         TimeUnit.MILLISECONDS)
      // 定期更新 recovery-point-offset-checkpoint 文件
      scheduler.schedule("kafka-recovery-point-checkpoint",
                         checkpointRecoveryPointOffsets,
                         delay = InitialTaskDelayMs,
```

```
                    period = flushCheckpointMs,
                    TimeUnit.MILLISECONDS)
    }
    // 日志合并，保留最新的数据
    if(cleanerConfig.enableCleaner)
      cleaner.startup()
  }
}
```

步骤 1　初始化 logs。Kafka 的日志目录结构如下：

```
drwxrwxr-x 2 hadoop hadoop 4096 Mar  8 11:47 Mytopic-0
drwxrwxr-x 2 hadoop hadoop 4096 Mar  8 11:47 Mytopic-3
drwxrwxr-x 2 hadoop hadoop 4096 Mar  8 11:47 Mytopic-4
drwxrwxr-x 2 hadoop hadoop 4096 Mar  8 11:47 Mytopic-6
drwxrwxr-x 2 hadoop hadoop 4096 Mar  8 11:47 Mytopic-8
drwxrwxr-x 2 hadoop hadoop 4096 Mar  8 11:47 Mytopic-9
-rw-rw-r-- 1 hadoop hadoop  356 Apr  6 10:45 recovery-point-offset-checkpoint
-rw-rw-r-- 1 hadoop hadoop  356 Apr  6 10:46 replication-offset-checkpoint
```

在 Broker Server 的每个日志目录下都有一份对应的 recovery-point-offset-checkpoint 文件，其日志目录由参数 log.dirs 决定，可以配置单个，也可以配置多个。这个 recovery-point-offset-checkpoint 文件记录了该目录下所有日志文件的状态，即 TopicAndPartition 和 Offset 的对应关系。通过这个文件来初始化 logs，也就是所谓的 Pool[TopicAndPartition, Log]() 对象。

步骤 2　开启后台定时任务和维护线程来管理日志。第一个定时任务是 cleanupLogs，该定时任务负责删除过时的数据和冗余的数据，实现如下：

```
def cleanupLogs() {
  debug("Beginning log cleanup...")
  var total = 0
  val startMs = time.milliseconds
  for(log <- allLogs; if !log.config.compact) {
    debug("Garbage collecting '" + log.name + "'")
    // 删除过期的数据和冗余的数据
    total += cleanupExpiredSegments(log) + cleanupSegmentsToMaintainSize(log)
  }
}
private def cleanupExpiredSegments(log: Log): Int = {
  val startMs = time.milliseconds
  /* log.config.retentionMs 由 log.retention.ms, log.retention.minutes 和
     log.retention.hours 决定 */
  log.deleteOldSegments(startMs - _.lastModified > log.config.retentionMs)
}
private def cleanupSegmentsToMaintainSize(log: Log): Int = {
  if(log.config.retentionSize < 0 || log.size < log.config.retentionSize)
    return 0
  // log.config.retentionSize 由 log.retention.bytes 决定
  var diff = log.size - log.config.retentionSize
  def shouldDelete(segment: LogSegment) = {
```

```
      if(diff - segment.size >= 0) {
        diff -= segment.size
        true
      } else {
        false
      }
    }
    log.deleteOldSegments(shouldDelete)
  }
```

以上删除数据的最小单位是 LogSegment。Kafka 通过配置时间和大小来实现消息的循环覆盖功能，比方说如果消息被配置为保留一天，那么，消息就会在磁盘保留一天的时间，也就是说，一天以内，任意消费这个消息。一天以后，这个消息就会被删除。保留多少时间就取决于业务和磁盘的大小。

第二个定时任务是 flushDirtyLogs，该定时任务负责刷新脏数据，刷新数据的最小单位是 LogSegment，LogSegment 里面包含数据文件和索引文件。实现如下：

```
  private def flushDirtyLogs() = {
    for ((topicAndPartition, log) <- logs) {
      try {
        // log.lastFlushTime 由 log.flush.interval.ms 决定
        val timeSinceLastFlush = time.milliseconds - log.lastFlushTime
        if(timeSinceLastFlush >= log.config.flushMs)
          log.flush
      } catch {
        case e: Throwable =>
          error("Error flushing topic " + topicAndPartition.topic, e)
      }
    }
  }
```

log.flush.interval.ms 决定了消息在内存中保留的时间，只有把数据刷新到磁盘才可以实现消息的持久化，否则如果遇到服务器宕机等异常情况会造成数据丢失。

第三个定时任务是 checkpointRecoveryPointOffsets，该定时任务负责定时刷新 logs 包含的元数据至 recovery-point-offset-checkpoint 文件，即 TopicAndPartition 和偏移量。因为 LogSegment 会间隔一定的时间不断地产生，此时需要及时地记录 LogSegment 的变化。实现如下：

```
  def checkpointRecoveryPointOffsets() {
    this.logDirs.foreach(checkpointLogsInDir)
  }
  private def checkpointLogsInDir(dir: File): Unit = {
    val recoveryPoints = this.logsByDir.get(dir.toString)
    if (recoveryPoints.isDefined) {
      // 写入的是 Map[TopicAndPartition, Long] 信息
      this.recoveryPointCheckpoints(dir).write(recoveryPoints.get.mapValues(_.
        recoveryPoint))
    }
  }
```

后台维护线程为日志合并线程，由参数 log.cleaner.enable 决定。Kafka 发送消息的时候需要携带 3 个参数，分别为 Topic、Key、Message，其中 Topic 为消息的主题，Key 为这条消息的分区值，Kafka 根据 Key 值决定这条消息落入哪个 Partition，Message 为消息的具体内容。当同一个 Key 值的不同 Message 出现多次，如果需要保留最新 Key 值对应的 Message 时，则需要开启日志合并线程，针对相同的 Key 值的不同 Message 只保留最后一个 Key 值对应的消息内容，其大致实现如图 4-9 所示。

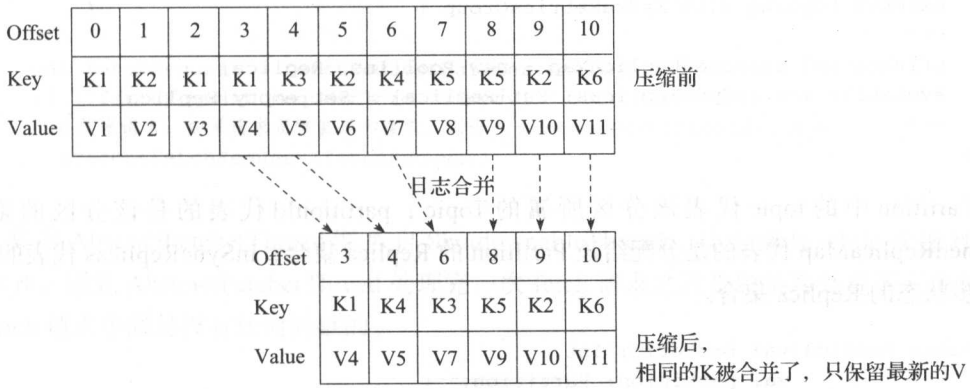


图 4-9　Kafka 的日志合并

4.3.2　ReplicaManager

ReplicaManager 负责提供针对 Topic 的分区副本数据的同步功能，需要针对不同的变化及时做出响应，例如 Partition 的 Replicas 发生 Leader 切换的时候，Partition 的 Replicas 所在的 Broker Server 离线的时候，Partition 的 Replicas 发生 Follower 同步 Leader 数据异常的时候，等等。

在继续深入之前首先来解释 2 个名词：AR 和 ISR。

- AR 是 Assign Replicas 的缩写，代表已经分配给 Partition 的副本。
- ISR 是 In-Sync Replicas 的缩写，代表处于同步状态的副本。

并不是所有的 AR 都是 ISR，尤其是当 Broker Server 离线的时候会导致对应 TopicAndPartition 的 Replica 没有及时同步 Leader 状态的 Replica，从而该 Replica 不是 ISR。执行 Kafka 提供的脚步查看 Topic 的元数据信息如图 4-10 所示。

```
[root@localhost bin]# kafka-topics.sh --describe --zookeeper localhost/kafka
Topic:my-topic    PartitionCount:4   ReplicationFactor:2   Configs:
    Topic: my-topic   Partition: 0   Leader: 1      Replicas: 1,4  Isr: 1,4
    Topic: my-topic   Partition: 1   Leader: 2      Replicas: 2,1  Isr: 2,1
    Topic: my-topic   Partition: 2   Leader: 3      Replicas: 3,2  Isr: 3,2
    Topic: my-topic   Partition: 3   Leader: 4      Replicas: 4,3  Isr: 4,3
```

图 4-10　Topic 的元数据信息

可见 Kafka 内部会维护 Topic 的状态，包括 Partition 的数目，Partition 的 Leader Replica、Partition 的 Replicas 和 Partition 的 ISR 等。其中 Partition 和 Replica 在 Kafka 源码中的结构大致如下：

```
class Partition(val topic: String,
                val partitionId: Int,
                time: Time,
                replicaManager: ReplicaManager)
  extends Logging with KafkaMetricsGroup {
  ......
  private val assignedReplicaMap = new Pool[Int, Replica]
  @volatile var inSyncReplicas: Set[Replica] = Set.empty[Replica]
  ......
}
```

Partition 中的 topic 代表该分区所属的 Topic；partitionId 代表的是该分区的索引；assignedReplicaMap 代表的是分配给该 Partition 的 Replica 集合；inSyncReplicas 代表的是处于同步状态的 Replica 集合。

```
class Replica(val brokerId: Int,
              val partition: Partition,
              time: Time = SystemTime,
              initialHighWatermarkValue: Long = 0L,
              val log: Option[Log] = None) extends Logging {
}
```

Replica 中的 brokerId 代表该 Replica 的位置，即位于哪个 Broker Server 上；partition 代表的是该 replica 所对应的 Partition，log 代表的是该 Replica 的日志数据和索引数据。

那么 ReplicaManager 是如何实现 Replica 数据的同步呢？主要利用 ReplicaFetcherThread（副本数据拉取线程）和 High Watermark Mechanism（高水位线机制）来实现数据的同步管理。

ReplicaFetcherThread 是位于 Broker Server 上的线程，线程个数是由 num.replica.fetchers 决定的。单个 ReplicaFetcherThread 线程只负责某个 Broker Server 上的部分 TopicAndPartition 的 Replica 数据同步，单个 ReplicaFetcherThread 线程不会负责跨多个 Broker Server 上的 TopicAndPartition 的 Replica 数据同步，即如图 4-11 所示。

当 Broker Server-0 上的 num.replica.fetchers=1 时，则此时针对其他 Broker Server 就分别有 num.replica.fetchers 个线程负责对应 Broker Server 上的数据拉取。增大 num.replica.fetchers 的个数可以增加 Follower 的 IO 并发度，及时同步 replica 的 Leader 和 Follower 的数据。

且看 AbstractFetcherThread 的处理流程：

```
override def doWork() {
  inLock(partitionMapLock) {
```

```
    if (partitionMap.isEmpty)
      partitionMapCond.await(200L, TimeUnit.MILLISECONDS)
      /*partitionMap 存放了该 fetch 线程所负责拉取的 TopicAndPartition，因此遍历
        partitionMap 可以生成对应的 fetchrequest*/
      partitionMap.foreach {
        case((topicAndPartition, offset)) =>
          fetchRequestBuilder.addFetch(topicAndPartition.topic, topicAndPartition.
            partition,
                                offset, fetchSize)
      }
  }
  val fetchRequest = fetchRequestBuilder.build()
  if (!fetchRequest.requestInfo.isEmpty)
    //处理请求，如果请求正常发送和接收，则进入 processPartitionData 流程
    processFetchRequest(fetchRequest)
}
```

其中 AbstractFetcherThread 继承自 ShutdownableThread，其内部是 while 死循环调用 doWork，因此 AbstractFetcherThread 处理完一次 fetch 请求之后是紧接着处理下一次的，两次 fetch 请求中间是没有任何间歇的。

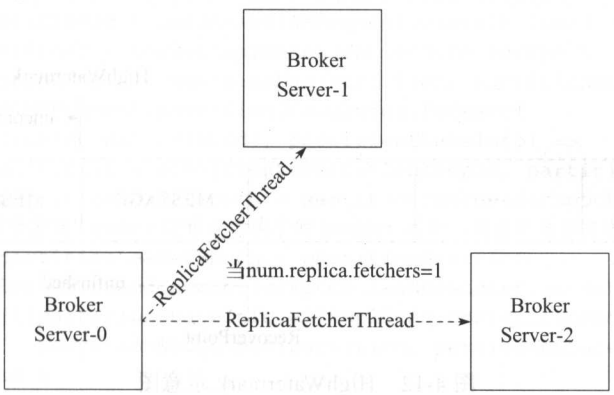


图 4-11　ReplicaFetcherThread

由于 ReplicaFetcherThread 继承自 AbstractFetcherThread，其 processPartitionData 的处理流程是在 ReplicaFetcherThread 里面实现的，即如下所示：

```
def processPartitionData(topicAndPartition: TopicAndPartition,
                         fetchOffset: Long,
                         partitionData: FetchResponsePartitionData) {
  try {
    //获取相应的 topic、partition、replica 元数据和 messageSet 真实数据
    val topic = topicAndPartition.topic
    val partitionId = topicAndPartition.partition
    val replica = replicaMgr.getReplica(topic, partitionId).get
    val messageSet = partitionData.messages.asInstanceOf[ByteBufferMessageSet]
```

```
        if (fetchOffset != replica.logEndOffset.messageOffset)
          throw new RuntimeException("Offset mismatch: fetched offset = %d, log end
           offset = %d.".format(fetchOffset, replica.logEndOffset.messageOffset))
          // 将 messageSet 写入 log
          replica.log.get.append(messageSet, assignOffsets = false)
          val followerHighWatermark = replica.logEndOffset.messageOffset.min
            (partitionData.hw)
          // 更新当前 replica 的 highWatermark
          replica.highWatermark = new LogOffsetMetadata(followerHighWatermark)
      } catch {
        case e: KafkaStorageException =>
          fatal("Disk error while replicating data.", e)
          Runtime.getRuntime.halt(1)
      }
    }
```

在这上面流程中最重的两步就是：1）将拉取的消息写入 log；2）更新当前 replica 的 HighWatermark。那什么是 HighWatermark 呢？其字面意思是高水位线，本质上代表的是 ISR 中所有 replicas 的 last commited message 的最小起始偏移量，即在这偏移之前的数据都被 ISR 中的所有 replicas 所接收，但是在这偏移之后的数据被 ISR 中的部分 replicas 所接收，如图 4-12 所示。

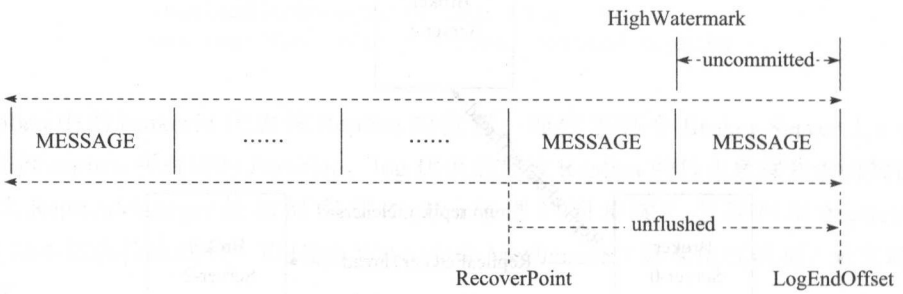


图 4-12 HighWatermark 示意图

其中 RecoverPoint 代表的是 recovery-point-offset-checkpoint 文件中记录的偏移量，LogEndOffset 代表的是当前 TopicAndPartition 的 replica（状态为 Leader）所接收到消息的最大偏移量，HighWatermark 代表的是已经同步给所有 ISR 的最小偏移量。Replica 的 HighWatermark 的更新发生在以下两种情况：1）Leader 状态的 Replica 接收到其他 Follower 状态的 Replica 的 FetchRequest 请求时，会选择性地更新 HighWatermark，其具体更新细节请参见 4.3.5.2FetchRequest 小节；2）Follower 状态的 Replica 接收到来自 Leader 状态的 Replica 的 FetchResponse 时，会选择性地更新 HighWatermark，即 ReplicaFetcherThread 内部的 processPartitionData 流程。

当某个 Broker Server 上被分配到 Replica 的时候会进入 becomeLeaderOrFollower 处理流程；当 Replica 被删除或者所在的 Broker Server 离线的时候会进入 stopReplicas 处理

流程；当 Follower 状态的 Replica 长时间没有同步 Leader 状态的 Replica 的时候会进入 maybeShrinkIsr 处理流程。

4.3.2.1　becomeLeaderOrFollower 处理流程

当 Broker Server 被分配 Repclica 的时候，该 Replica 有可能成为 Leader 状态的 Replica 或者 Follower 状态的 Replica，也有可能发生两者状态的切换，此时就会进入 become-LeaderOrFollower 处理流程，代码如下所示：

```
def becomeLeaderOrFollower(leaderAndISRRequest: LeaderAndIsrRequest,
                           offsetManager: OffsetManager)
  : (collection.Map[(String, Int), Short], Short) = {
  ......
  replicaStateChangeLock synchronized {
    val responseMap = new collection.mutable.HashMap[(String, Int), Short]
    /* 判断 request 的时效性，如果 request 内部的时钟小于当前的时钟，则这条 request 就已经过时了 */
    if(leaderAndISRRequest.controllerEpoch < controllerEpoch) {
      ......
      (responseMap, ErrorMapping.StaleControllerEpochCode)
    } else {
      val controllerId = leaderAndISRRequest.controllerId
      val correlationId = leaderAndISRRequest.correlationId
      controllerEpoch = leaderAndISRRequest.controllerEpoch
      val partitionState = new HashMap[Partition, PartitionStateInfo]()
      leaderAndISRRequest.partitionStateInfos.foreach{
        case ((topic, partitionId), partitionStateInfo) =>
          val partition = getOrCreatePartition(topic, partitionId)
          val partitionLeaderEpoch = partition.getLeaderEpoch()
          // 如果当前的 leader 时钟小于请求的 leader 时钟，则说明当前的状态是有效的
          if (partitionLeaderEpoch < partitionStateInfo.
              leaderIsrAndControllerEpoch.leaderAndIsr.leaderEpoch) {
              if(partitionStateInfo.allReplicas.contains(config.brokerId))
                 partitionState.put(partition, partitionStateInfo)
            else {
              ......
            }
          } else {
            ......
            responseMap.put((topic, partitionId), ErrorMapping.StaleLeaderEpochCode)
          }
      }
      // 筛选出该 Broker Server 上即将成为 Leader 的 Replica
      val partitionsTobeLeader = partitionState.filter{
        case (partition, partitionStateInfo) =>
          partitionStateInfo.leaderIsrAndControllerEpoch.leaderAndIsr.leader ==
            config.brokerId
      }
      // 筛选出该 Broker Server 上即将成为 Follower 的 Replica
      val partitionsToBeFollower = (partitionState -- partitionsTobeLeader.keys)
      if (!partitionsTobeLeader.isEmpty)
```

```
        //进入成为 Leader 的流程
        makeLeaders(controllerId, controllerEpoch, partitionsTobeLeader,
                    leaderAndISRRequest.correlationId, responseMap, offsetManager)
      if (!partitionsToBeFollower.isEmpty)
        //进入成为 Follower 的流程
        makeFollowers(controllerId, controllerEpoch,
                      partitionsToBeFollower, leaderAndISRRequest.leaders,
                      leaderAndISRRequest.correlationId, responseMap, offsetManager)
      if (!hwThreadInitialized) {
        /* 开启 highwatermark-checkpoint 线程，该线程负责将 HighWatermark 刷新
           replication-offset-checkpoint 文件 */
        startHighWaterMarksCheckPointThread()
        hwThreadInitialized = true
      }
      //关闭空闲的 Fetch 线程
      replicaFetcherManager.shutdownIdleFetcherThreads()
      (responseMap, ErrorMapping.NoError)
    }
  }
}
```

becomeLeaderOrFollower 流程中最重要的还是针对不同的 Replica 状态进行处理，当 Replica 为 Leader 时，其处理流程如下：

```
private def makeLeaders(controllerId: Int,
                        epoch: Int,
                        partitionState: Map[Partition, PartitionStateInfo],
                        correlationId: Int,
                        responseMap: mutable.Map[(String, Int), Short],
                        offsetManager: OffsetManager) = {
  ......
  try {
    ......
    //如果是 leader 的话，必然要删除针对该 Replica 的 Fetch 请求
    replicaFetcherManager.removeFetcherForPartitions(partitionState.keySet.
      map(new TopicAndPartition(_)))
    ......
    //调用 Partition 的 makeLeader 流程
    partitionState.foreach{ case (partition, partitionStateInfo) =>
      partition.makeLeader(controllerId, partitionStateInfo, correlationId,
        offsetManager)}
  } catch {
    ......
  }
  ......
}
```

最终还是通过 Partition 的 makeLeader 来完成 Leader 状态的 Replica 初始化，即：

```
def makeLeader(controllerId: Int,
```

```
                partitionStateInfo: PartitionStateInfo,
                correlationId: Int,
                offsetManager: OffsetManager): Boolean = {
  inWriteLock(leaderIsrUpdateLock) {
    val allReplicas = partitionStateInfo.allReplicas
    val leaderIsrAndControllerEpoch = partitionStateInfo.leaderIsrAndControllerEpoch
    val leaderAndIsr = leaderIsrAndControllerEpoch.leaderAndIsr
    controllerEpoch = leaderIsrAndControllerEpoch.controllerEpoch
    // 尝试生成 AR (assign replicas) 的结构化信息，不存在则生成；存在则获取
    allReplicas.foreach(replica => getOrCreateReplica(replica))
    val newInSyncReplicas = leaderAndIsr.isr.map(r => getOrCreateReplica(r)).toSet
    // 删除已经不存在的 Replica
      (assignedReplicas().map(_.brokerId) -- allReplicas).foreach(removeReplica(_))
    inSyncReplicas = newInSyncReplicas
    leaderEpoch = leaderAndIsr.leaderEpoch
    zkVersion = leaderAndIsr.zkVersion
    leaderReplicaIdOpt = Some(localBrokerId)
    val newLeaderReplica = getReplica().get
    // 初始化 Leader 状态的 Replica 的 LogOffsetMetadata
    newLeaderReplica.convertHWToLocalOffsetMetadata()
    // 初始化 Follower 状态的 Replica 的 LogOffsetMetadata
    assignedReplicas.foreach(r => if (r.brokerId != localBrokerId) r.logEndOffset =
      LogOffsetMetadata.UnknownOffsetMetadata)
    // 初始化 Leader 状态的 Replica 的 highWatermark
    maybeIncrementLeaderHW(newLeaderReplica)
    ......
    true
  }
}
```

当 Replica 为 Follower 时，其处理流程如下：

```
private def makeFollowers(controllerId: Int,
                          epoch: Int,
                          partitionState: Map[Partition, PartitionStateInfo],
                          leaders: Set[Broker],
                          correlationId: Int,
                          responseMap: mutable.Map[(String, Int), Short],
                          offsetManager: OffsetManager) {
  ......
  try {
    var partitionsToMakeFollower: Set[Partition] = Set()
    partitionState.foreach{ case (partition, partitionStateInfo) =>
      val leaderIsrAndControllerEpoch = partitionStateInfo.leaderIsrAndControllerEpoch
      val newLeaderBrokerId = leaderIsrAndControllerEpoch.leaderAndIsr.leader
      // 只改变那些其 Leader Broker 处于在线状态的分区
      leaders.find(_.id == newLeaderBrokerId) match {
        case Some(leaderBroker) =>
          // 调用 Partition 的 makeFollower 流程
          if (partition.makeFollower(controllerId, partitionStateInfo,
              correlationId, offsetManager))
```

```
                partitionsToMakeFollower += partition
            else
              ......
          case None =>
            ......
        }
      }
      // 由于 leader 可能发生变化，则删除旧的 Fetch 请求
      replicaFetcherManager.removeFetcherForPartitions(partitionsToMakeFollower.
        map(new TopicAndPartition(_)))
      ......
      /* 为了保证所有 Replica 的数据一致性，需要将该 Replica 的数据截断至其 highWatermark 处，
         可以参考图 4-12*/
      logManager.truncateTo(partitionsToMakeFollower.map(partition => (
        new TopicAndPartition(partition),
        partition.getOrCreateReplica().highWatermark.messageOffset)).toMap)
      if (isShuttingDown.get()) {
        ......
      }
      else {
        val partitionsToMakeFollowerWithLeaderAndOffset =
          partitionsToMakeFollower.map( partition =>
            new TopicAndPartition(partition) -> BrokerAndInitialOffset(
              leaders.find(_.id == partition.leaderReplicaIdOpt.get).get,
            partition.getReplica().get.logEndOffset.messageOffset)).toMap
        // 增加新的 Fetch 请求
        replicaFetcherManager.addFetcherForPartitions(
        partitionsToMakeFollowerWithLeaderAndOffset)
        ......
      }
    } catch {
      ......
    }
  }
```

同样，最终还是通过 Partition 的 makeFollower 来完成 Follower 状态的 Replica 初始化，如下所示：

```
  def makeFollower(controllerId: Int,
                   partitionStateInfo: PartitionStateInfo,
                   correlationId: Int,
                   offsetManager: OffsetManager): Boolean = {
    inWriteLock(leaderIsrUpdateLock) {
      val allReplicas = partitionStateInfo.allReplicas
      val leaderIsrAndControllerEpoch = partitionStateInfo.leaderIsrAndControllerEpoch
      val leaderAndIsr = leaderIsrAndControllerEpoch.leaderAndIsr
      val newLeaderBrokerId: Int = leaderAndIsr.leader
      controllerEpoch = leaderIsrAndControllerEpoch.controllerEpoch
      // 增加新的 Replica
      allReplicas.foreach(r => getOrCreateReplica(r))
```

```
        // 删除旧的 Replica
        (assignedReplicas().map(_.brokerId) -- allReplicas).foreach(removeReplica(_))
        inSyncReplicas = Set.empty[Replica]
        leaderEpoch = leaderAndIsr.leaderEpoch
        zkVersion = leaderAndIsr.zkVersion
        ......
        /* 如果该 Partition 的 Leader Replica 没有发生变化，则表明其对应的 Replica 保持不变；
           否则表明对应的 Replica 发生变化 */
        if (leaderReplicaIdOpt.isDefined && leaderReplicaIdOpt.get == newLeaderBrokerId) {
          false
        } else {
          leaderReplicaIdOpt = Some(newLeaderBrokerId)
          true
        }
      }
    }
```

4.3.2.2　stopReplicas 处理流程

当 Replica 被删除或者所在的 Broker Server 离线时会进入 stopReplicas 处理流程，如下所示：

```
  def stopReplicas(stopReplicaRequest: StopReplicaRequest)
    : (mutable.Map[TopicAndPartition, Short], Short) = {
    replicaStateChangeLock synchronized {
      val responseMap = new collection.mutable.HashMap[TopicAndPartition, Short]
      /* 判断 request 的时效性，如果 request 内部的时钟小于当前的时钟，则这条 request 已经过时了 */
      if(stopReplicaRequest.controllerEpoch < controllerEpoch) {
        ......
        (responseMap, ErrorMapping.StaleControllerEpochCode)
      } else {
        controllerEpoch = stopReplicaRequest.controllerEpoch
        // 删除旧的 Fetch 请求
        replicaFetcherManager.removeFetcherForPartitions(
          stopReplicaRequest.partitions.map(r => TopicAndPartition(r.topic, r.partition)))
        for(topicAndPartition <- stopReplicaRequest.partitions){
          // 根据 deletePartitions 判断是否删除该 Replica
          val errorCode = stopReplica(topicAndPartition.topic,
                                      topicAndPartition.partition,
                                      stopReplicaRequest.deletePartitions)
          responseMap.put(topicAndPartition, errorCode)
        }
        (responseMap, ErrorMapping.NoError)
      }
    }
  }
```

可见 stopReplicas 流程相对简单，首先无论如何都会删除旧的 Fetch 请求，其次根据 deletePartitions 判断是否删除该 Replica，如果是的话，则删除 Replica（包括 log 和 index 数据）。

4.3.2.3 maybeShrinkIsr 处理流程

ReplicaManager 启动的时候会开启 isr-expiration 线程。因为在 Replica 数据实时同步过程中，如果某些 Broker Server 由于异常原因没有及时同步其相应的 Replica 数据，则位于这些 Broker Server 上的 Replica 需要从 ISR 列表中删除，其处理流程如下：

```
def maybeShrinkIsr(replicaMaxLagTimeMs: Long,
                   replicaMaxLagMessages: Long) {
  inWriteLock(leaderIsrUpdateLock) {
    // 只有当 Leader 的 Replica 位于该 Broker Server 上时才进行 ShrinkIsr 流程
    leaderReplicaIfLocal() match {
      case Some(leaderReplica) =>
        // 获取没有及时同步的 Replica 列表
        val outOfSyncReplicas = getOutOfSyncReplicas(leaderReplica,
          replicaMaxLagTimeMs, replicaMaxLagMessages)
        if(outOfSyncReplicas.size > 0) {
          // 获取当前新的 ISR
          val newInSyncReplicas = inSyncReplicas -- outOfSyncReplicas
          assert(newInSyncReplicas.size > 0)
          // 更新 ISR 至 ZK
          updateIsr(newInSyncReplicas)
          // 更新该 Partition 的 highWatermark
          maybeIncrementLeaderHW(leaderReplica)
          replicaManager.isrShrinkRate.mark()
        }
      case None =>
    }
  }
}
```

maybeShrinkIsr 仅仅是将更新之后的 ISR 写入 ZK，同时维护该 Partition 的 HighWatermark。那么如何识别同步异常的 Replica 呢？ Kafka 中主要用到以下两个参数：

- replica.lag.time.max.ms，此参数限制了 Follower 状态下的 Replica 和 Leader 状态下的 Replica 进行同步的最大时间间隔，如果超过最大时间间隔，则说明 Follower 状态下的 Replica 同步已经异常，需要从 ISR 中删除。
- replica.lag.max.messages，此参数限制了 Follower 状态下的 Replica 允许落后 Leader 状态下的 Replica 的最大消息条数，如果超过最大消息条数，则说明 Follower 状态下的 Replica 同步已经异常，需要从 ISR 中删除。

其具体的识别流程如下所示：

```
def getOutOfSyncReplicas(leaderReplica: Replica,
                         keepInSyncTimeMs: Long,
                         keepInSyncMessages: Long): Set[Replica] = {
  val leaderLogEndOffset = leaderReplica.logEndOffset
  // 把 Leader 从 ISR 中删除
  val candidateReplicas = inSyncReplicas - leaderReplica
  // 根据 replica.lag.time.max.ms 判断
  val stuckReplicas = candidateReplicas.filter(r => (time.milliseconds -
```

```
    r.logEndOffsetUpdateTimeMs) > keepInSyncTimeMs)
  ......
  // 根据 replica.lag.max.messages 判断
  val slowReplicas = candidateReplicas.filter(r =>
    r.logEndOffset.messageOffset >= 0 &&
    leaderLogEndOffset.messageOffset - r.logEndOffset.messageOffset >
      keepInSyncMessages)
  ......
  // 将两者结果叠加返回
  stuckReplicas ++ slowReplicas
}
```

4.3.3　OffsetManager

Kafka 提供两种保存 Consumer 偏移量的方法：

1）将偏移量保存至 Zookeeper 中。

2）将偏移量保存至 Kafka 内部一个名为 __consumer_offsets 的 Topic 里面。将偏移量保存至 Zookeeper 中是 Kafka 一直就支持的，但是考虑到 Zookeeper 并不适合大批量的频繁写入操作，因此 Kafka 开始支持将 Consumer 的偏移量保存在 Kafka 内部的 Topic 中，即 __consumer_offsets Topic。当用户配置 offsets.storage = kafka 时，高级消费者会将偏移量保存至 Topic 里面，同时通过 OffsetManager 提供对这些偏移量的管理。

OffsetManager 主要提供以下几个功能：

- 缓存最新的偏移量。
- 提供对偏移量的查询。
- Compact，保留最新的偏移量，以此来控制该 Topic 的日志大小。

OffsetManager 的基本组成如下：

```
class OffsetManager(val config: OffsetManagerConfig,
                    replicaManager: ReplicaManager,
                    zkClient: ZkClient,
                    scheduler: Scheduler) extends Logging with KafkaMetricsGroup {
  private val offsetsCache = new Pool[GroupTopicPartition, OffsetAndMetadata]
  ......
  scheduler.schedule(name = "offsets-cache-compactor",
                     fun = compact,
                     period = config.offsetsRetentionCheckIntervalMs,
                     unit = TimeUnit.MILLISECONDS)
  ......
}
```

其中 offsetsCache 提供针对 Consumer 偏移量的保存和查询，compact 作为定时任务，间隔 config.offsetsRetentionCheckIntervalMs 执行。

4.3.3.1　Consumer 偏移量的保存

Consumer 将偏移量保存至内部名为 __consumer_offsets 的 Topic 里面类似于生产者将

消息发布到 Kafka 集群的 Topic 里面，即将偏移量包装成消息发送至 __consumer_offsets。当 Broker Server 接收到消息时，除了将消息保存至日志之外，还会调用 OffsetManager 提供的 putOffsets 方法将消息保存至 offsetsCache 中（其 Broker Server 接收生产消息请求的具体细节可参考 4.3.5.1 小节），如下所示：

```
def putOffsets(group: String,
               offsets: Map[TopicAndPartition, OffsetAndMetadata]) {
  offsets.foreach {
    case (topicAndPartition, offsetAndMetadata) =>
      // 将 group, topicAndPartition 组装成 offsetsCache map 中的 key
      putOffset(GroupTopicPartition(group, topicAndPartition), offsetAndMetadata)
  }
}
```

4.3.3.2 Consumer 偏移量的读取

当 Broker Server 接收到查询偏移量的请求时，如果发现偏移量保存在 Kafka 中时，则调用 OffsetManager 提供的 getOffsets 方法将偏移量取出（其 Broker Server 接收查询偏移量请求的具体细节可参考 4.3.5.10 小节），如下所示：

```
def getOffsets(group: String,
               topicPartitions: Seq[TopicAndPartition])
  : Map[TopicAndPartition, OffsetMetadataAndError] = {
  // 计算该 group 位于 __consumer_offsets 的哪个分区
  val offsetsPartition = partitionFor(group)
  followerTransitionLock synchronized {
    // 只有 partition 为 leader 所在的 Broker Server 提供查询服务
    if (leaderIsLocal(offsetsPartition)) {
      /* 如果目标 partition 的数据正在加载，则无法获取其偏移量，其只会发生在 Broker Server
         启动阶段，因为需要从指定的主分区加载数据 */
      if (loadingPartitions synchronized loadingPartitions.contains(offsetsPartition)) {
        topicPartitions.map { topicAndPartition =>
          val groupTopicPartition = GroupTopicPartition(group, topicAndPartition)
          (groupTopicPartition.topicPartition, OffsetMetadataAndError.OffsetsLoading)
        }.toMap
      } else {
        // 如果 topicPartitions 的大小为 0，则获取该 group 的所有偏移量信息
        if (topicPartitions.size == 0) {
          offsetsCache.filter(_._1.group == group).map {
            case(groupTopicPartition, offsetAndMetadata) =>
              (groupTopicPartition.topicPartition,
               OffsetMetadataAndError(offsetAndMetadata.offset,
                                      offsetAndMetadata.metadata,
                                      ErrorMapping.NoError))
          }.toMap
        } else {
          /* 如果 topicPartitions 的大小不为 0，则获取该 group 特定的 topicAndPartition
             的偏移量信息 */
          topicPartitions.map {
```

```
                topicAndPartition =>
                  val groupTopicPartition = GroupTopicPartition(group, topicAndPartition)
                  (groupTopicPartition.topicPartition, getOffset(groupTopicPartition))
              }.toMap
            }
          }
        } else {
          // partition 不为 leader，不对外提供服务
          topicPartitions.map {
            topicAndPartition =>
              val groupTopicPartition = GroupTopicPartition(group, topicAndPartition)
              (groupTopicPartition.topicPartition, OffsetMetadataAndError.
               NotOffsetManagerForGroup)
          }.toMap
        }
      }
    }
```

其中需要注意的一点是：Kafka 是如何将 Consumer Group 产生的偏移量信息保存在 __consumer_offsets 的不同分区上的？其本质是通过计算不同 Consumer Group 的 hash 值和 __consumer_offsets 的分区数的模数，其结果作为指定分区的索引。因此在 getOffsets 的第一步就开始进行取模运算，即：

```
// 取 group 的 hash 绝对值和 offsetsTopicNumPartitions 的模数
def partitionFor(group: String): Int =
  Utils.abs(group.hashCode) % config.offsetsTopicNumPartitions
```

不同 group 对应的偏移量信息在 __consumer_offsets 的 Topic 里面存放的格式如图 4-13 所示。

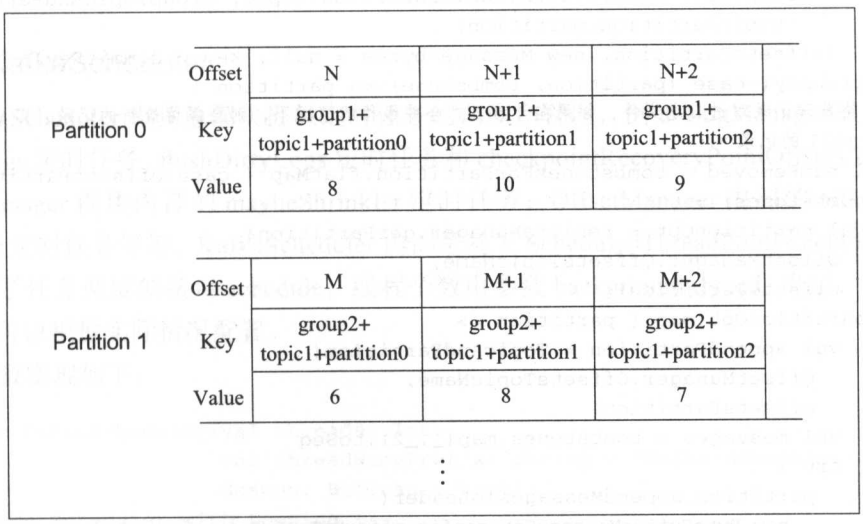


图 4-13　不同 group 对应的偏移量信息在 __consumer_offsets 中的存放格式

Consumer Group 本身决定了其所拥有的偏移量信息位于哪个 Partition 上；Group、Topic 和 Partition 三者决定了 Partition 上的 Key 值，其内容为具体的偏移量。

4.3.3.3 Compact 策略

当 Consumer Group 经过长时间运行之后，不再产生偏移量信息时，很可能其已经不需要保存在 __consumer_offsets 里面的偏移量信息了，此时 Broker Server 需要有一种机制去清理之前保存的偏移量，这就是所谓的 Compact 策略。这个策略会将长时间没有更新的 Consumer Group 对应的偏移量清理掉，保留持续不断在更新的 Consumer Group 的偏移量，其具体的执行过程如下：

```
private def compact() {
  val startMs = SystemTime.milliseconds
  /* 通过将当前时间减去上次更新的时候判断 offsetsCache 里面每个 Key 的活跃程度，将那些长时间
     没有更新的筛选出来 */
  val staleOffsets = offsetsCache.filter(startMs - _._2.timestamp > config.
    offsetsRetentionMs)
  /* tombstone 在英语中为墓碑的意思，也就是死亡的意思，即那些 Consumer Group 可能已经不会
     再消费记录了 */
  val tombstonesForPartition = staleOffsets.map {
    case(groupTopicAndPartition, offsetAndMetadata) =>
      /* 根据 Consumer  Group 的 hash 值和 __consumer_offsets 的分区数的模数筛选出特定的
         分区索引 */
      val offsetsPartition = partitionFor(groupTopicAndPartition.group)
      // 先从内存中删除
      offsetsCache.remove(groupTopicAndPartition)
      /* 然后产生墓碑记录，即 bytes 为 null，只是将 Key 传进去产生空记录，最后按 partition 分组 */
      val commitKey = OffsetManager.offsetCommitKey(
        groupTopicAndPartition.group,
        groupTopicAndPartition.topicPartition.topic, groupTopicAndPartition.
          topicPartition.partition)
      (offsetsPartition, new Message(bytes = null, key = commitKey))
  }.groupBy{ case (partition, tombstone) => partition }
  /* 将墓碑记录写进日志文件，如果在开启日志合并线程的情况下，则会保留最新的记录，即 Value 为
     null 的记录 */
  val numRemoved = tombstonesForPartition.flatMap { case(offsetsPartition,
    tombstones) =>
    val partitionOpt = replicaManager.getPartition(
      OffsetManager.OffsetsTopicName,
      offsetsPartition)
    partitionOpt.map { partition =>
      val appendPartition = TopicAndPartition(
        OffsetManager.OffsetsTopicName,
        offsetsPartition)
      val messages = tombstones.map(_._2).toSeq
      try {
        partition.appendMessagesToLeader(
          new ByteBufferMessageSet(config.offsetsTopicCompressionCodec, messages:_*))
        tombstones.size
```

```
        }
        catch {
          case t: Throwable =>
            error("Failed to mark %d stale offsets for deletion in %s.".format
              (messages.size, appendPartition), t)
            0
        }
      }
    }.sum
  }
```

为什么要写 bytes=null 的空值呢？因为在开启日志合并线程（参考 4.3.1.3 小节）的情况下，最后只会保留为 null 的记录，即如图 4-14 所示。

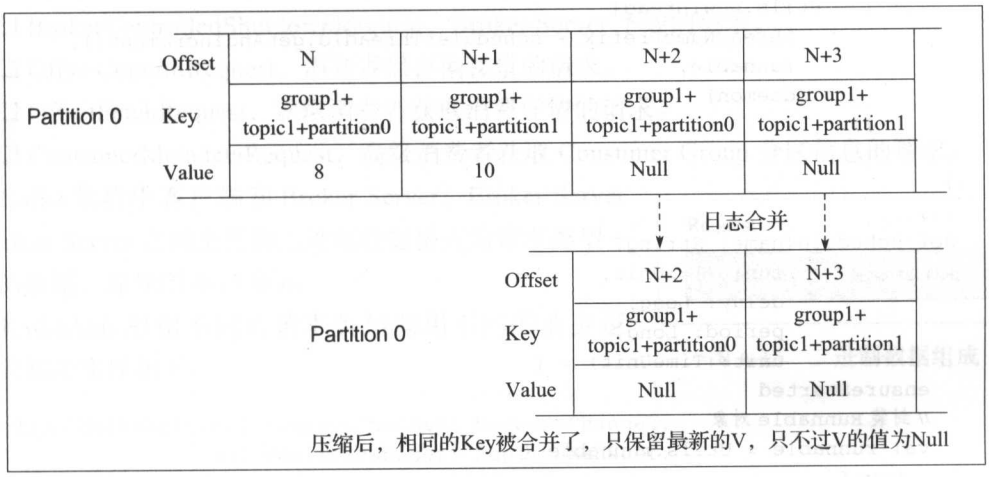


图 4-14　Compact 策略

4.3.4　KafkaScheduler

KafkaScheduler 为其他模块提供定时任务的调度和管理，例如 LogManager 模块内部的 cleanupLogs 定时任务，flushDirtyLogs 定时任务和 checkpointRecoveryPointOffsets 定时任务；ReplicaManager 模块内部的 maybeShrinkIsr 定时任务；OffsetManager 内部的 offsets-cache-compactor 定时任务等等。KafkaScheduler 内部是基于 ScheduledThreadPoolExecutor 实现的，对外封装了任务调度的接口 schedule，线程个数由参数 background.threads 决定，默认值为 10，用户可以根据实际情况配置。

其大致实现如下：

```
class KafkaScheduler(val threads: Int,
                     val threadNamePrefix: String = "kafka-scheduler-",
                     daemon: Boolean = true)
extends Scheduler with Logging {
  @volatile private var executor: ScheduledThreadPoolExecutor = null
```

```
override def startup() {
  this synchronized {
    if(executor != null)
      throw new IllegalStateException("This scheduler has already been started!")
    /* 基于 ScheduledThreadPoolExecutor 实现，提供了比 Timer/TimerTask 更好的任务
       管理和异常机制 */
    executor = new ScheduledThreadPoolExecutor(threads)
    /* 当在 shutdown 的时候，取消任何已经超时的任务的执行，包括正在执行的和正在排队的 */
    executor.setContinueExistingPeriodicTasksAfterShutdownPolicy(false)
    executor.setExecuteExistingDelayedTasksAfterShutdownPolicy(false)
    // 线程工厂类
    executor.setThreadFactory(
      new ThreadFactory() {
        def newThread(runnable: Runnable): Thread =
          Utils.newThread(
            threadNamePrefix + schedulerThreadId.getAndIncrement(),
            runnable,
            daemon)
      })
  }
}
......
def schedule(name: String,
             fun: ()=>Unit,
             delay: Long,
             period: Long,
             unit: TimeUnit) = {
  ensureStarted
  // 封装 Runnable 对象
  val runnable = Utils.runnable {
    try {
      fun()
    } catch {
      case t: Throwable => error("Uncaught exception in scheduled task '" +
        name +"'", t)
    } finally {
      trace("Completed execution of scheduled task '%s'.".format(name))
    }
  }
  // 如果是周期性任务，则周期性执行；否则到期执行
  if(period >= 0)
    executor.scheduleAtFixedRate(runnable, delay, period, unit)
  else
    executor.schedule(runnable, delay, unit)
}
}
```

4.3.5 KafkaApis

KafkaApis 模块主要负责不同业务请求的具体实现逻辑。需要利用以下四个模块相互配

合来完成操作：LogManager、ReplicaManager、OffsetManager、KafkaScheduler。这四个模块已经在前序章节中讲解完毕，本节主要讲解 KafkaApis 内部针对不同请求的实现细节。

KafkaApis 负责的请求类型包括以下十一类：

- ProducerRequest，生产者发送消息的请求。
- FetchRequest，消费者获取消息的请求。
- OffsetRequest，获取 Topic 当前 offset 的元数据信息的请求。
- TopicMetadataRequest，获取 Topic 元数据信息的请求。
- LeaderAndIsrRequest，Topic 的元数据信息发生变化的请求。
- StopReplicaRequest，停止拷贝副本数据的请求。
- UpdateMetadataRequest，更新 Topic 元数据信息的请求。
- BrokerControlledShutdownRequest，Broker Server 下线的请求。
- OffsetCommitRequest，消费者保存偏移量的请求。
- OffsetFetchRequest，获取消费者获取消费详情的请求。
- ConsumerMetadataRequest，高级消费者获取 Consumer Group 分区信息的请求。

Kafka 集群中客户端和 Broker Server、Broker Server 和 Broker Server 之间交互的二进制数据格式为请求类型 + 请求的数据，即如图 4-15 所示。

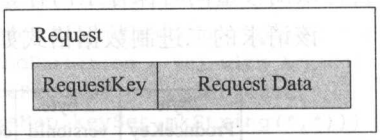


图 4-15　二进制数据组成

KafkaApis 根据不同的请求类型调用不同的业务逻辑，其初步实现如下：

```
class KafkaApis(val requestChannel: RequestChannel,
                val replicaManager: ReplicaManager,
                val offsetManager: OffsetManager,
                val zkClient: ZkClient,
                val brokerId: Int,
                val config: KafkaConfig,
                val controller: KafkaController) extends Logging {
  ......
  def handle(request: RequestChannel.Request) {
    try{
      // 根据不同的 RequestKeys 匹配不同的业务实现逻辑
      request.requestId match {
        case RequestKeys.ProduceKey => handleProducerOrOffsetCommitRequest(request)
        case RequestKeys.FetchKey => handleFetchRequest(request)
        case RequestKeys.OffsetsKey => handleOffsetRequest(request)
        case RequestKeys.MetadataKey => handleTopicMetadataRequest(request)
        case RequestKeys.LeaderAndIsrKey => handleLeaderAndIsrRequest(request)
        case RequestKeys.StopReplicaKey => handleStopReplicaRequest(request)
        case RequestKeys.UpdateMetadataKey => handleUpdateMetadataRequest(request)
        case RequestKeys.ControlledShutdownKey => handleControlledShutdownRequest
          (request)
        case RequestKeys.OffsetCommitKey => handleOffsetCommitRequest(request)
```

```
            case RequestKeys.OffsetFetchKey => handleOffsetFetchRequest(request)
            case RequestKeys.ConsumerMetadataKey => handleConsumerMetadataRequest(request)
            case requestId => throw new KafkaException("Unknown api code " + requestId)
          }
        } catch {
          case e: Throwable =>
            request.requestObj.handleError(e, requestChannel, request)
            error("error when handling request %s".format(request.requestObj), e)
        } finally
          request.apiLocalCompleteTimeMs = SystemTime.milliseconds
      }
    }
```

4.3.5.1 ProducerRequest

当生产者发送消息保存至 Kafka 集群时或者高级消费者发送偏移量保存至 Kafka 集群时，都会发送此种类型的请求，对于后者可以将其看成一种特殊的消息，消息里面的内容是特定 Topic 的偏移量。当 Broker Server 收到此种类型的消息时，主要完成以下两件事情：1）持久化消息；2）组装响应格式。除此之外，如果是高级消费者发送的偏移量请求时，则会将该偏移量同时保存至内存，以便支持后续的快速读取。

该请求的二进制数据格式如图 4-16 所示。

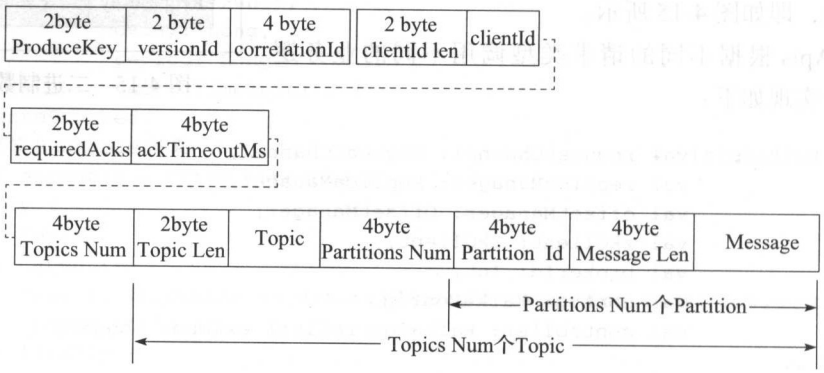


图 4-16 ProducerRequest 二进制数据组成

其详细实现过程如下：

```
def handleProducerOrOffsetCommitRequest(request: RequestChannel.Request) {
  val (produceRequest, offsetCommitRequestOpt) =
    if (request.requestId == RequestKeys.OffsetCommitKey) {
      //如果是高级消费者发送的，则需要将其包装成 ProducerRequest
      val offsetCommitRequest = request.requestObj.asInstanceOf[OffsetCommitRequest]
      OffsetCommitRequest.changeInvalidTimeToCurrentTime(offsetCommitRequest)
      (producerRequestFromOffsetCommit(offsetCommitRequest), Some(offsetCommitRequest))
    } else {
      (request.requestObj.asInstanceOf[ProducerRequest], None)
    }
```

```
  //校验 ack 的有效性，ack 只能为 -1,0,1，大于 1 将会废除
  if (produceRequest.requiredAcks > 1 || produceRequest.requiredAcks < -1) {
      warn(("Client %s from %s sent a produce request with request.required.
            acks of %d, which is now deprecated and will " +
            "be removed in next release. Valid values are -1, 0 or 1. Please
            consult Kafka documentation for supported " +
            "and recommended configuration.").format(produceRequest.clientId,
            request.remoteAddress, produceRequest.requiredAcks))
  }
val sTime = SystemTime.milliseconds
//将消息持久化
val localProduceResults = appendToLocalLog(produceRequest, offsetCommitRequestOpt.
  nonEmpty)
//过滤异常错误
val firstErrorCode = localProduceResults.find(_.errorCode != ErrorMapping.NoError)
                  .map(_.errorCode).getOrElse(ErrorMapping.NoError)
//统计出现错误的 Partition 个数
val numPartitionsInError = localProduceResults.count(_.error.isDefined)
if(produceRequest.requiredAcks == 0) {
  if (numPartitionsInError != 0) {
    //当 ack 为 0 时，如果出现持久化消息异常，则主动关闭链接
    info(("Send the close connection response due to error handling produce request " +
         "[clientId = %s, correlationId = %s, topicAndPartition = %s] with Ack=0")
         .format(produceRequest.clientId, produceRequest.correlationId,
         produceRequest.topicPartitionMessageSizeMap.keySet.mkString(",")))
    requestChannel.closeConnection(request.processor, request)
  } else {
    if (firstErrorCode == ErrorMapping.NoError)
      //没有任何错误，则将高级消费者发送的偏移量保存至 offsetManager 的内存中
      offsetCommitRequestOpt.foreach(
        ocr => offsetManager.putOffsets(ocr.groupId, ocr.requestInfo))
    if (offsetCommitRequestOpt.isDefined) {
      /* 如果是高级消费者发送的，那么即使 ack=0，也需要将消息的持久化结果返回给高级消费者 */
      val response = offsetCommitRequestOpt.get.responseFor(
        firstErrorCode,
        config.offsetMetadataMaxSize)
      requestChannel.sendResponse(
        new RequestChannel.Response(request,
                                    new BoundedByteBufferSend(response)))
    } else
      /* ack=0，客户端不关心服务端的具体执行情况，只关心服务端是否收到请求，因此不需要返回
         详细的执行结果 */
      requestChannel.noOperation(request.processor, request)
  }
}
else if (produceRequest.requiredAcks == 1 ||
      produceRequest.numPartitions <= 0 ||
      numPartitionsInError == produceRequest.numPartitions) {
  /*ack=1 或者目标分区个数无效或者持久化全部失败的情况下，需要返回具体的执行结果 */
  if (firstErrorCode == ErrorMapping.NoError) {
    //没有任何错误，则将高级消费者发送的偏移量保存至 offsetManager 的内存中
```

```
        offsetCommitRequestOpt.foreach(
          ocr => offsetManager.putOffsets(ocr.groupId, ocr.requestInfo))
      }
      //将请求的具体执行结果返回给客户端，无论客户端是生产者还是高级消费者
      val statuses = localProduceResults.map(
        r => r.key -> ProducerResponseStatus(r.errorCode, r.start)).toMap
      val response = offsetCommitRequestOpt.map(_.responseFor(
        firstErrorCode,
        config.offsetMetadataMaxSize)).getOrElse(
          ProducerResponse(produceRequest.correlationId, statuses))
      requestChannel.sendResponse(new RequestChannel.Response(
        request,
        new BoundedByteBufferSend(response)))
    } else{
      //ack=-1，需要等待（min.insync.replicas-1）个副本同步数据之后才返回响应
      val producerRequestKeys = produceRequest.data.keys.toSeq
      val statuses = localProduceResults.map(r =>
        r.key -> DelayedProduceResponseStatus(
          r.end + 1,
          ProducerResponseStatus(r.errorCode, r.start))).toMap
      //从字面意思就是产生一个延迟的请求，不会立即返回响应
      val delayedRequest = new DelayedProduce(producerRequestKeys,
                                              request,
                                              produceRequest.ackTimeoutMs.toLong,
                                              produceRequest,
                                              statuses,
                                              offsetCommitRequestOpt)
      //判断该请求是否满足返回的要求，如果满足，则返回相应的响应，否则延迟返回响应
      val satisfiedByMe = producerRequestPurgatory.checkAndMaybeWatch(delayedRequest)
      if (satisfiedByMe)
          producerRequestPurgatory.respond(delayedRequest)
    }
    produceRequest.emptyData()
  }
```

ProducerRequest.requiredAcks 的取值是由 request.required.acks 决定的，在返回响应的时候主要针对 request.required.acks 的不同取值进行不同的处理：

- 当 request.required.acks 等于 0 时，生产者不关心 Broker Server 端消息持久化的执行结果，Broker Server 只需要简单的返回即可，但是对于高级消费者发送的提交偏移量的请求还是需要返回具体的执行结果。
- 当 request.required.acks 等于 1 时，无论是生产者发送的请求还是高级消费者发送的情况，都需要将 Broker Server 端消息持久化的执行结果立刻返回给对应的客户端。
- 当 request.required.acks 等于 -1 时，此时不会立刻返回 Broker Server 端消息持久化的结果，而是需要等待 Partition 的 ISR 列表中的 Replica 完成数据同步，且 ISR 列表的个数大于 min.insync.replicas 时才会将响应返回给对应的客户端

针对 request.required.acks 等于 0 或者等于 1 时的场景比较好理解，那么当 request.required.

acks=-1 时，Kafka 是如何做到延迟答复的呢？它采用的是一种称为 Purgatory 的策略，接下来将主要讲解这种策略的实现原理。

首先我们来看 DelayedProduce 的组成：

```
class DelayedProduce(override val keys: Seq[TopicAndPartition],
                     override val request: RequestChannel.Request,
                     override val delayMs: Long,
                     val produce: ProducerRequest,
                     val partitionStatus: Map[TopicAndPartition,
                     DelayedProduceResponseStatus],
                     val offsetCommitRequestOpt: Option[OffsetCommitRequest] = None)
  extends DelayedRequest(keys, request, delayMs) with Logging {
  ......
}
```

DelayedProduce 类内部包含了一个称为 keys 的变量，针对当前 ProducerRequest 中不同的 TopicAndPartition 对 DelayedProduce 进行分类。

其次我们来看一下 RequestPurgatory 的组成：

```
abstract class RequestPurgatory[T <: DelayedRequest](
  brokerId: Int = 0,
  purgeInterval: Int = 1000)
  extends Logging with KafkaMetricsGroup {
  /* 针对 DelayedProduce 内部不同的 TopicAndPartition 产生对应的 Watchers，Watchers 内
     部又保存对应的 DelayedProduce*/
  private val watchersForKey = new Pool[Any, Watchers](Some((key: Any) => new Watchers))
  /* 检测超时请求线程，如果该请求超过一定时间还没达到满足条件，则无论如何都将该请求的响应返回
     给客户端 */
  private val expiredRequestReaper = new ExpiredRequestReaper
  private val expirationThread = Utils.newThread(name="request-expiration-task",
                                                 runnable=expiredRequestReaper,
                                                 daemon=false)
}
```

RequestPurgatory 内部的 watchersForKey 保存了不同的 TopicAndPartition 对应的 Watchers，同时 Watchers 内部的 requests 保存了相同 TopicAndPartition 对应的 DelayedProduce，即相互之间的关系如图 4-17 所示。

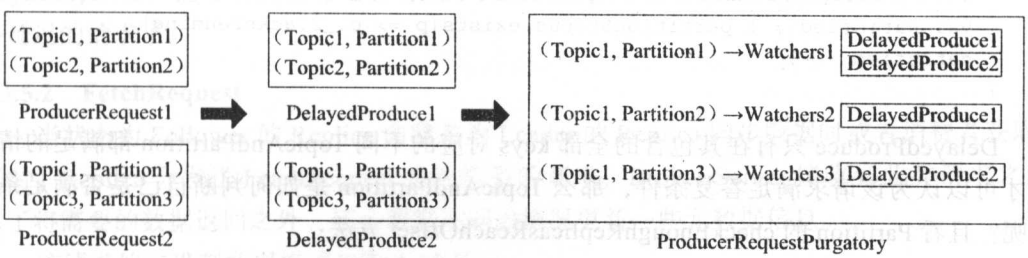


图 4-17　watchersForKey 和 Watcher 的关系

接着我们来看一下什么时候会触发 Broker Server 回复 DelayedProduce 的响应。只有当该 Broker Server 上对应 Partition 的 HighWatermark 发生改变时，Broker Server 才会去检查是否满足条件，如果满足则返回响应，如果不满足则继续等待，直到超时。Partition 的 HighWatermark 反应了其对应分区所有 Replica 的数据一致性情况，只会在 Partition 的 ISR 发生变化和 Broker Server 接收到对应的 FetchRequest 请求时发生改变。那么如何判断是否满足条件呢？且看 DelayedProduce 的 isSatisfied 方法：

```
def isSatisfied(replicaManager: ReplicaManager) = {
  partitionStatus.foreach { case (topicAndPartition, fetchPartitionStatus) =>
    // 判断每个分区的数据是否在等待其他 Broker Server 来获取
    if (fetchPartitionStatus.acksPending) {
      // 获取对应的 Partition 对象
      val partitionOpt = replicaManager.getPartition(topicAndPartition.topic,
                                                     topicAndPartition.partition)
      // 检查该 Partition 是否有足够的副本达到了指定的偏移量
      val (hasEnough, errorCode) = partitionOpt match {
        case Some(partition) =>
          partition.checkEnoughReplicasReachOffset(
            fetchPartitionStatus.requiredOffset,
            produce.requiredAcks)
        case None =>
          (false, ErrorMapping.UnknownTopicOrPartitionCode)
      }
      /* 如果出现异常，比方说 Leader Replica 不在此 Broker Server 上，则不需要其他 Broker
       Server 来进行同步 */
      if (errorCode != ErrorMapping.NoError) {
          fetchPartitionStatus.acksPending = false
          fetchPartitionStatus.responseStatus.error = errorCode
      } else if (hasEnough) {
        // 否则只有在 hasEnough 为 true 的时候设置 acksPending 为 false
        fetchPartitionStatus.acksPending = false
        fetchPartitionStatus.responseStatus.error = ErrorMapping.NoError
      }
    }
  }
  // 只有该 DelayedProduce 所有的 keys 都满足的时候才认为满足
  val satisfied = ! partitionStatus.exists(p => p._2.acksPending)
  satisfied
}
```

DelayedProduce 只有在其包含的全部 keys 对应的不同 TopicAndPartition 都满足的情况下才可以认为该请求满足答复条件，那么 TopicAndPartition 是如何判断自己是否满足条件的呢？且看 Partition 的 checkEnoughReplicasReachOffset 方法：

```
def checkEnoughReplicasReachOffset(
```

```
    requiredOffset: Long,
    requiredAcks: Int): (Boolean, Short) = {
    leaderReplicaIfLocal() match {
    case Some(leaderReplica) =>
      //获取 ISR 列表
      val curInSyncReplicas = inSyncReplicas
      //统计 ISR 中每个 Replica 的记录偏移量是否大于指定的大小
      val numAcks = curInSyncReplicas.count(r => {
        if (!r.isLocal)
          r.logEndOffset.messageOffset >= requiredOffset
        else
          true
      })
      //获取 min.insync.replicas 的设置
      val minIsr = leaderReplica.log.get.config.minInSyncReplicas
      //如果 requiredAcks 小于 0，且 Leader Replica 的 HW 大于指定大小
      if (requiredAcks < 0 && leaderReplica.highWatermark.messageOffset >=
        requiredOffset ) {
        //判断 minIsr 是否小于当前 ISR
        if (minIsr <= curInSyncReplicas.size) {
          (true, ErrorMapping.NoError)
        } else {
          (true, ErrorMapping.NotEnoughReplicasAfterAppendCode)
        }
      } else if (requiredAcks > 0 && numAcks >= requiredAcks) {
        //requiredAcks 不会大于 0，因为只有小于 0 才会采用延迟答复
        (true, ErrorMapping.NoError)
      } else
        (false, ErrorMapping.NoError)
    case None =>
      (false, ErrorMapping.NotLeaderForPartitionCode)
    }
  }
```

Partition 主要判断 Leader Replica 的 HighWatermark，只要其大于指定的大小，则说明 ISR 列表的所有 Replica 都 Fetch 到足够的数据了，并且 ISR 列表的个数大于指定的 min.insync.replicas，就可以说明有足够的副本同步到数据了，那么 Broker Server 就可以回复这些延迟的请求了。

4.3.5.2　FetchRequest

当状态为 Follower 的 Replica 向状态为 Leader 的 Replica 同步数据时或者消费者获取数据时，Replica 会发送 FetchRequest 此种类型的请求，Broker Server 接收到此类请求之后，除了将需要的数据返回之外，还会根据不同的情况更新一些元数据信息。

该请求的二进制数据格式如图 4-18 所示。

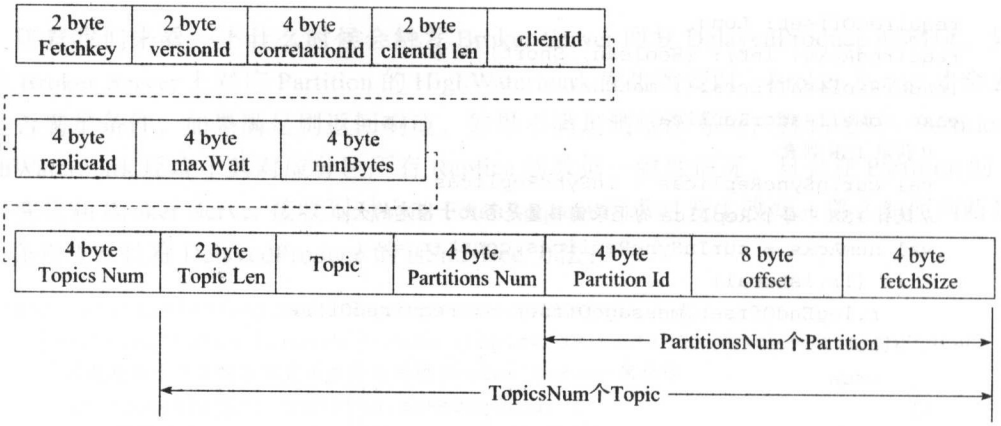


图 4-18 FetchRequest 二进制数据组成

其详细实现过程如下：

```
def handleFetchRequest(request: RequestChannel.Request) {
  val fetchRequest = request.requestObj.asInstanceOf[FetchRequest]
  //根据 fetchRequest 获取指定的数据
  val dataRead = replicaManager.readMessageSets(fetchRequest)
  /* 如果是来自 Follower 的 Replica 的请求，则还需要额外更新元数据信息以及响应那些延迟的
     DelayedProduce 和 DelayedFetch*/
  if(fetchRequest.isFromFollower)
    recordFollowerLogEndOffsets(fetchRequest.replicaId, dataRead.mapValues(_.offset))
    //统计获取的数据量
  val bytesReadable = dataRead.values.map(_.data.messages.sizeInBytes).sum
  //统计异常
  val errorReadingData = dataRead.values.foldLeft(false)((errorIncurred,
    dataAndOffset) =>
    errorIncurred || (dataAndOffset.data.error != ErrorMapping.NoError))
  /*
   * 如果发生以下几种情况，则立刻返回响应：
   *1）发送方不希望等待，只想立刻返回
   *2）fetchRequest 不需要获取数据
   *3）已经获取足够的数据
   *4）获取数据出现异常
   */
  if(fetchRequest.maxWait <= 0 ||
    fetchRequest.numPartitions <= 0 ||
    bytesReadable >= fetchRequest.minBytes ||
    errorReadingData) {
    val response = new FetchResponse(fetchRequest.correlationId, dataRead.
      mapValues(_.data))
    requestChannel.sendResponse(new RequestChannel.Response(
      request,
      new FetchResponseSend(response)))
  } else {
```

```
        /* 只有当没有获取到足够多的数据时，此时不会立刻返回响应，而是采用 Purgatory 策略延迟响应 */
        val delayedFetchKeys = fetchRequest.requestInfo.keys.toSeq
        val delayedFetch = new DelayedFetch(delayedFetchKeys,
                                            request,
                                            fetchRequest.maxWait,
                                            fetchRequest,dataRead.mapValues(_.offset))
        val satisfiedByMe = fetchRequestPurgatory.checkAndMaybeWatch(delayedFetch)
        if (satisfiedByMe)
          fetchRequestPurgatory.respond(delayedFetch)
      }
    }
```

对于 Broker Server 处理 FetchRequest 请求的过程中有两点需要注意：1）如果是来自状态为 Follower 的 Replica 的请求，则如何更新元数据信息；2）DelayedFetch 是如何判断满足条件的。

对于前者主要更新以下几个方面的元数据：1）更新对应 Replica 的 LEO(LogEndOffset)，即记录该 Replica 的当前日志结束偏移量；2）更新 Leader Replica 的 highWatermark；3）可能需要扩大 ISR 列表，是因为发生了数据同步，导致可能有更多的 Replica 已经和状态为 Leader 的 Replica 保持了同步。更新完数据之后，就去检查该 Broker Server 上的 DelayedProduce 是否满足条件返回响应，因为发生了数据同步之后，导致之前的 DelayedProduceRequest 得到了更多 Replica 的确认，其大致过程如下：

```
  private def recordFollowerLogEndOffsets(replicaId: Int,
                                          offsets: Map[TopicAndPartition,
                                          LogOffsetMetadata]) {
    offsets.foreach {
      case (topicAndPartition, offset) =>
        /* 更新对应 Replica 的 LEO，更新 Leader Replica 的 highWatermark，更新 ISR，
           如果 Leader Replica 的 highWatermark 发生变化，则 unblock 之前阻塞住的
           DelayedFetchRequest 和 DelayedProduceRequest*/
        replicaManager.updateReplicaLEOAndPartitionHW(topicAndPartition.topic,
                                                      topicAndPartition.partition,
                                                      replicaId,
                                                      offset)
        // 针对 DelayedProduceRequest 的 ack>1 的情况
        replicaManager.unblockDelayedProduceRequests(topicAndPartition)
    }
  }
```

对于后者主要是判断当前是否有足够的数据可以提供拉取，如果有足够的数据可以拉取，则响应那些 DelayedFetchRequest，其详细过程如下：

```
  def isSatisfied(replicaManager: ReplicaManager) : Boolean = {
    var accumulatedSize = 0
    val fromFollower = fetch.isFromFollower
    partitionFetchOffsets.foreach {
      case (topicAndPartition, fetchOffset) =>
```

```
        try {
          if (fetchOffset != LogOffsetMetadata.UnknownOffsetMetadata) {
            //获取 Leader Replica
            val replica = replicaManager.getLeaderReplicaIfLocal(topicAndPartition.
                          topic,topicAndPartition.partition)
            val endOffset =
              /*
                1) 如果是来自 Follower，则可以同步 Leader Replica 的 logEndOffset 之前的
                数据，即所有的数据
                2) 如果是来自消费者的，则只能同步 Leader Replica 的 highWatermark 之前的
                数据，即所有 Replica 都包含的数据
               */
              if (fromFollower)
                replica.logEndOffset
              else
                replica.highWatermark
            if (endOffset.offsetOnOlderSegment(fetchOffset)) {
              /* 此时代表 Follower Replica 正在向被截断的 Leader Replica 拉取数据，则需要
                立刻返回 Response*/
              return true
            } else if (fetchOffset.offsetOnOlderSegment(endOffset)) {
              /* 此时代表 Follower Replica 正在拉取 Leader Replica 旧的 Segment 上的数据，
                说明 Follower Replica 落后太多，则需要立刻返回 Response*/
              return true
            } else if (fetchOffset.precedes(endOffset)) {
              //在当前 Segment 上，并且 fetchOffset< endOffset，说明有数据可以获取
              accumulatedSize += endOffset.positionDiff(fetchOffset)
            }
          }
        } catch {
          case utpe: UnknownTopicOrPartitionException =>
            return true
          case nle: NotLeaderForPartitionException =>
            return true
        }
      }
      //统计可以拉取的数据量是否满足条件
      accumulatedSize >= fetch.minBytes
    }
```

在 Kafka 集群正常运行过程中，fetchOffset 和 endOffset 都位于当前的 Segment 上，因此主要是统计可以 Fetch 的数据量是否满足最小数据量的要求，只要满足，就可以响应那些 DelayedFetchRequest。

4.3.5.3 OffsetRequest

当消费者或者是客户端想要获取 Topic 某个时间段内的偏移量详情时会发送此种类型的请求，Broker Server 接收到此请求之后，会返回指定修改时间之前的 LogSegment 的 baseOffset。

该请求的二进制数据格式如图 4-19 所示。

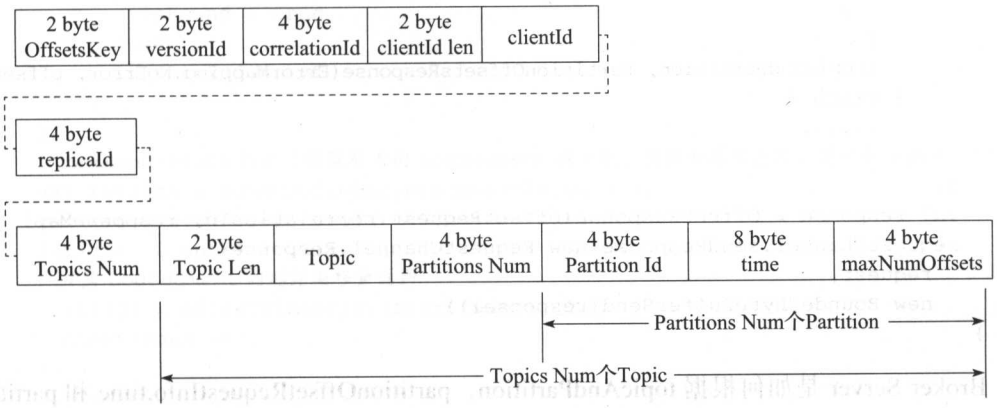


图 4-19　OffsetRequest 二进制数据组成

其详细实现过程如下：

```
def handleOffsetRequest(request: RequestChannel.Request) {
  val offsetRequest = request.requestObj.asInstanceOf[OffsetRequest]
  val responseMap = offsetRequest.requestInfo.map(elem => {
    val (topicAndPartition, partitionOffsetRequestInfo) = elem
    try {
      val localReplica =
        if(!offsetRequest.isFromDebuggingClient)
          //请求来自于普通消费者，则 Leader 的 Replica 必须位于此 Broker Server 上
          replicaManager.getLeaderReplicaIfLocal(topicAndPartition.topic,
                                                 topicAndPartition.partition)
        else
          /*仅仅是为了调试，则不需要关心该 Broker Server 是否是指定的 topicAndPartition
            的 Leader Replica*/
          replicaManager.getReplicaOrException(topicAndPartition.topic,
                                               topicAndPartition.partition)
        val offsets = {
          //根据 topicAndPartition, partitionOffsetRequestInfo.time 和
            partitionOffsetRequestInfo.maxNumOffsets 获取偏移量详情
          val allOffsets = fetchOffsets(replicaManager.logManager,
                                        topicAndPartition,
                                        partitionOffsetRequestInfo.time,
                                        partitionOffsetRequestInfo.maxNumOffsets)
          if (!offsetRequest.isFromOrdinaryClient) {
            //请求不是来自于消费者（调试目的），则可以返回所有的偏移量详情
            allOffsets
          } else {
            //请求来自于消费者，则只能返回 hw 以下的偏移量详情
            val hw = localReplica.highWatermark.messageOffset
            if (allOffsets.exists(_ > hw))
              hw +: allOffsets.dropWhile(_ > hw)
```

```
            else
              allOffsets
          }
        }
        (topicAndPartition, PartitionOffsetsResponse(ErrorMapping.NoError, offsets))
      } catch {
        ......
    }
  })
  val response = OffsetResponse(offsetRequest.correlationId, responseMap)
  requestChannel.sendResponse(new RequestChannel.Response(
    request,
    new BoundedByteBufferSend(response)))
}
```

Broker Server 是如何根据 topicAndPartition，partitionOffsetRequestInfo.time 和 partitionOffsetRequestInfo.maxNumOffsets 来获取 LogSegment 的起始偏移量的呢？在 fetchOffsets 内部最终会调用 fetchOffsetsBefore 函数，从 Before 字面意思就可以猜到是获取指定修改时间之前的 LogSegment 详细信息，其详细流程如下：

```
def fetchOffsetsBefore(log: Log,
                       timestamp: Long,
                       maxNumOffsets: Int): Seq[Long] = {
  val segsArray = log.logSegments.toArray
  var offsetTimeArray: Array[(Long, Long)] = null
  //判断最新的 LogSegment 是否有值，有值的话，则也需要考虑该 LogSegment
  if(segsArray.last.size > 0)
    offsetTimeArray = new Array[(Long, Long)](segsArray.length + 1)
  else
    offsetTimeArray = new Array[(Long, Long)](segsArray.length)
  /* 填充 offsetTimeArray 里面的值，其中 key 为 LogSegment 的起始偏移量，value 为
     LogSegment 的最后修改时间 */
  for(i <- 0 until segsArray.length)
    offsetTimeArray(i) = (segsArray(i).baseOffset, segsArray(i).lastModified)
  if(segsArray.last.size > 0)
    /* 需要注意的是针对最新的 LogSegment，其 key 为下一条消息的偏移量，value 为当前时间 */
    offsetTimeArray(segsArray.length) = (log.logEndOffset, SystemTime.milliseconds)
  var startIndex = -1
  timestamp match {
    case OffsetRequest.LatestTime =>
      //指定最新，则从最后往前返回
      startIndex = offsetTimeArray.length - 1
    case OffsetRequest.EarliestTime =>
      //指定最早，则从最前往前返回，即返回第一个元素
      startIndex = 0
    case _ =>
      /*timestamp 为其他值的情况下，从后往前查找最后修改时间小于当前值的那个元素位置 */
      var isFound = false
      startIndex = offsetTimeArray.length - 1
```

```
      while (startIndex >= 0 && !isFound) {
        if (offsetTimeArray(startIndex)._2 <= timestamp)
          isFound = true
        else
          startIndex -=1
      }
    }
    /*maxNumOffsets 指定了获取最大的 LogSegment 的个数，需要和结果比较，返回较小集合 */
    val retSize = maxNumOffsets.min(startIndex + 1)
    val ret = new Array[Long](retSize)
    for(j <- 0 until retSize) {
      // 将 LogSegment 的起始偏移量返回
      ret(j) = offsetTimeArray(startIndex)._1
      startIndex -= 1
    }
    ret.toSeq.sortBy(- _)
  }
```

4.3.5.4　TopicMetadataRequest

当生产者或者消费者想要获取 Topic 的元数据信息时会发送此种类型的请求，Broker Server 接收到此请求之后，会把当前在线的所有 Broker Server 和 Topic 的所有分区信息返回，后者包括分区索引、Leader Replica、Assign Replicas、In-Sync Replicas 等。

该请求的二进制数据格式如图 4-20 所示。

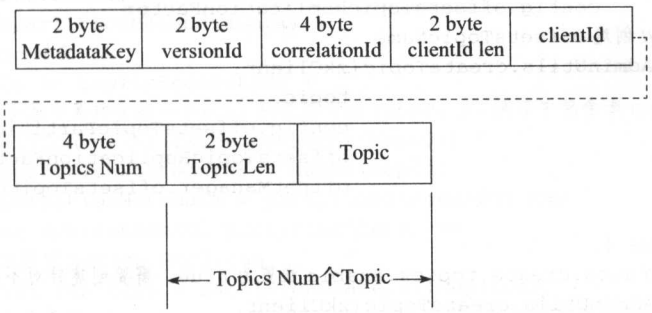


图 4-20　TopicMetadataRequest 二进制数据组成

其详细实现过程如下：

```
def handleTopicMetadataRequest(request: RequestChannel.Request) {
  val metadataRequest = request.requestObj.asInstanceOf[TopicMetadataRequest]
  // 指定 Topic 获取元数据信息
  val topicMetadata = getTopicMetadata(metadataRequest.topics.toSet)
  // 返回当前在线的 Broker Server 列表
  val brokers = metadataCache.getAliveBrokers
  // 将当前在线的 Broker Server 列表和指定 topic 的元数据信息返回
  val response = new TopicMetadataResponse(brokers,
                                           topicMetadata,
                                           metadataRequest.correlationId)
```

```
    requestChannel.sendResponse(new RequestChannel.Response(
      request,
      new BoundedByteBufferSend(response)))
  }
```

在这其中最重要的是如何返回 Topic 的元数据信息，且看 getTopicMetadata 的实现细节：

```
  private def getTopicMetadata(topics: Set[String]): Seq[TopicMetadata] = {
    // 从 metadataCache 获取正常创建的 Topic 的分区信息
    val topicResponses = metadataCache.getTopicMetadata(topics)
    if (topics.size > 0 && topicResponses.size != topics.size) {
      /* 此时表明有些 Topic 的分区信息没有获取到，这些 Topic 可能是一些特殊的 Topic，需要做特殊处理 */
      val nonExistentTopics = topics -- topicResponses.map(_.topic).toSet
      val responsesForNonExistentTopics = nonExistentTopics.map { topic =>
        if (topic == OffsetManager.OffsetsTopicName || config.autoCreateTopicsEnable) {
          try {
            if (topic == OffsetManager.OffsetsTopicName) {
              /* 如果是高级消费者保存偏移量的 Topic，则需要根据当前在线 Broker Server 个数和
                 offsetsTopicReplicationFactor 来创建，这种情况仅仅只会发生一次 */
              val aliveBrokers = metadataCache.getAliveBrokers
              val offsetsTopicReplicationFactor =
                if (aliveBrokers.length > 0)
                  Math.min(config.offsetsTopicReplicationFactor, aliveBrokers.length)
                else
                  config.offsetsTopicReplicationFactor
              // 创建 OffsetsTopicName
              AdminUtils.createTopic(zkClient,
                                     topic,
                                     config.offsetsTopicPartitions,
                                     offsetsTopicReplicationFactor,
                                     offsetManager.offsetsTopicConfig)
            }
            else {
              // auto.create.topics.enable 配置为 true，需要创建针对不存在的普通 Topic
              AdminUtils.createTopic(zkClient,
                                     topic,
                                     config.numPartitions,
                                     config.defaultReplicationFactor)
            }
          } catch {
            case e: TopicExistsException =>
          }
          /* 针对首次获取不存在的 Topic，即使后来重新创建了，其错误码还是 ErrorMapping.
             LeaderNotAvailableCode*/
          new TopicMetadata(topic,
                            Seq.empty[PartitionMetadata],
                            ErrorMapping.LeaderNotAvailableCode)
        } else {
          /* 如果既不是高级消费者保存偏移量的 Topic，也没有配置 auto.create.topics.
```

```
            enable 为 true，则这些 Topic 为不存在 */
          new TopicMetadata(topic,
                            Seq.empty[PartitionMetadata],
                            ErrorMapping.UnknownTopicOrPartitionCode)
        }
      }
      topicResponses.appendAll(responsesForNonExistentTopics)
    }
    topicResponses
  }
```

getTopicMetadata 除了从 metadataCache 获取正常 Topic 的元数据信息之外，针对那些不存在的 Topic 会根据不同的情况做不同的处理，尤其在 auto.create.topics.enable 配置为 true 的情况下，会自动创建那些不存在的 Topic，其分区个数由 default.replication.factor 配置参数决定，默认值为 1。getTopicMetadata 从 metadataCache 获取正常 Topic 的元数据信息流程如下：

```
def getTopicMetadata(topics: Set[String]) = {
  // 如果 topics 为空，则代表获取所有 Topic 的元数据信息
  val isAllTopics = topics.isEmpty
  // 提取具体的 Topic
  val topicsRequested = if(isAllTopics) cache.keySet else topics
  val topicResponses: mutable.ListBuffer[TopicMetadata] =
    new mutable.ListBuffer[TopicMetadata]
  inReadLock(partitionMetadataLock) {
    // 遍历 Topic 列表
    for (topic <- topicsRequested) {
      // 满足以下两个条件中的一个：1）获取所有的 Topic；2）缓存中包含该 topic
      if (isAllTopics || cache.contains(topic)) {
        val partitionStateInfos = cache(topic)
        val partitionMetadata = partitionStateInfos.map {
          case (partitionId, partitionState) =>
            // 获取 Assign Replicas
            val replicas = partitionState.allReplicas
            // 筛选出 Live Assign Replicas
            val replicaInfo: Seq[Broker] = replicas.map(
              aliveBrokers.getOrElse(_, null)).filter(_ != null).toSeq
            var leaderInfo: Option[Broker] = None
            var isrInfo: Seq[Broker] = Nil
            val leaderIsrAndEpoch = partitionState.leaderIsrAndControllerEpoch
            // 获取 Leader Replica
            val leader = leaderIsrAndEpoch.leaderAndIsr.leader
            // 获取 In-Sync Replicas
            val isr = leaderIsrAndEpoch.leaderAndIsr.isr
            val topicPartition = TopicAndPartition(topic, partitionId)
            try {
              // 获取 leader 的 Broker Server
              leaderInfo = aliveBrokers.get(leader)
              if (!leaderInfo.isDefined)
```

```
                    // 如果 leaderInfo 为空，则说明 Leader Replica 不存在
                    throw new LeaderNotAvailableException(
                      "Leader not available for %s".format(topicPartition))
                  // 获取 Live In-Sync Replicas
                  isrInfo = isr.map(aliveBrokers.getOrElse(_, null)).filter(_ != null)
                  if (replicaInfo.size < replicas.size)
                    /* 如果 Live Assign Replicas 没有满足初始副本因子，则说明当前在线副本个数不足 */
                    throw new ReplicaNotAvailableException("Replica information
                      not available for following brokers: " +
                      replicas.filterNot(replicaInfo.map(_.id).contains(_)).mkString(","))
                  if (isrInfo.size < isr.size)
                    /* 如果 Live In-Sync Replicas 没有满足 In-Sync Replicas，则说明
                     In-Sync Replicas 中有 Replica 下线 */
                    throw new ReplicaNotAvailableException("In Sync Replica
                      information not available for following brokers: " +
                      isr.filterNot(isrInfo.map(_.id).contains(_)).mkString(","))
                  // 正常情况下 PartitionMetadata 的 errorCode 为 ErrorMapping.NoError
                  new PartitionMetadata(partitionId,
                                        leaderInfo,
                                        replicaInfo,
                                        isrInfo,
                                        ErrorMapping.NoError)
                } catch {
                  case e: Throwable =>
                    new PartitionMetadata(partitionId, leaderInfo, replicaInfo, isrInfo,
                      ErrorMapping.codeFor(e.getClass.asInstanceOf[Class[Throwable]]))
                }
              }
              topicResponses += new TopicMetadata(topic, partitionMetadata.toSeq)
            }
          }
        }
        topicResponses
      }
```

4.3.5.5 LeaderAndIsrRequest

如果某 Broker Server 由于异常导致宕机，则原先 Leader Replica 在该 Broker Server 上的 Topic 会发生 Replica 的 Leader 切换以及 In-Sync Replicas 列表的缩减；如果由于网络延迟，导致原先位于 In-Sync Replicas 列表中的 Replica 没有长时间向 Leader Replica 同步数据，最后导致落后太多的数据，此时需要将该 Replica 从 ISR 列表中剔除；如果用户手动重新指定 Topic 的分布情况，则可能发生 Topic 的 Replica Leader 切换以及 ISR 列表的变化等等。

当发生以上几种情况时，Leader 状态的 KafkaController 所在的 Broker Server 会向相关 Broker Server 下发 Topic 的 Leader 或者 ISR 列表发生变化的请求，Broker Server 接收到此请求之后，会根据具体的内容对发生变化的 Replica 进行相应的处理。

该请求的二进制数据格式如图 4-21 所示。

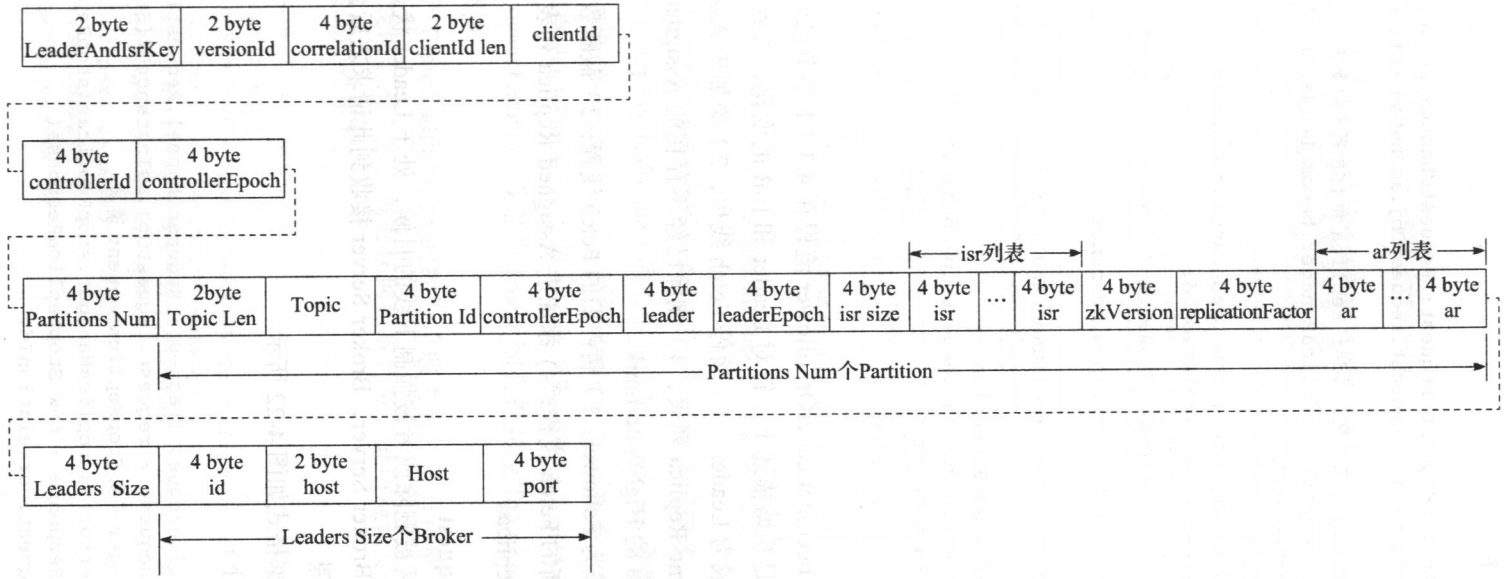


图 4-21　LeaderAndIsrRequest 二进制数据组成

其详细实现过程如下：

```
def handleLeaderAndIsrRequest(request: RequestChannel.Request) {
  val leaderAndIsrRequest = request.requestObj.asInstanceOf[LeaderAndIsrRequest]
  try {
    // 通过 ReplicaManager 来实现 Leader Replica 和 ISR 列表的变化
    val (response, error) = replicaManager.becomeLeaderOrFollower(
      leaderAndIsrRequest,
      offsetManager)
    val leaderAndIsrResponse = new LeaderAndIsrResponse(
      leaderAndIsrRequest.correlationId,
      response,
      error)
    requestChannel.sendResponse(new Response(
      request,
      new BoundedByteBufferSend(leaderAndIsrResponse)))
  } catch {
    case e: KafkaStorageException =>
      fatal("Disk error during leadership change.", e)
      Runtime.getRuntime.halt(1)
  }
}
```

ReplicaManager 的 becomeLeaderOrFollower 流程在 4.3.2.1 小节已经详细描述，读者可以参考此小节，本节只是粗略描述下。针对 Leader 和 ISR 的变化，大致分两种情况：

- 当某个 Replica 成为 Leader：1）暂停 Fetch 线程；2）添加进 Assigned Replica 列表；3）添加进 In-Sync Replica 列表；4）删除已经不存在的 Assigned Replica；5）初始化 Leader Replica 的 HighWatermark。
- 当某个 Replica 成为 Follower：1）暂停旧的 Fetch 线程；2）截断数据至 HighWatermark 以下；3）开启新的 Fetch 线程；4) 添加进 Assigned Replica 列表；5）删除已经不存在的 Assigned Replica。

4.3.5.6 StopReplicaRequest

当 Topic 的某个分区被删除或者被强制下线的时候，处于 Leader 状态的 KafkaController 会发送此请求至相关的 Broker Server，Broker Server 接收到此请求之后会针对 Topic 分区被删除的情况做相应的处理。

该请求的二进制数据格式如图 4-22 所示。

其详细实现过程如下：

```
def handleStopReplicaRequest(request: RequestChannel.Request) {
  val stopReplicaRequest = request.requestObj.asInstanceOf[StopReplicaRequest]
  // 通过 ReplicaManager 实现 StopReplicaRequest 请求
  val (response, error) = replicaManager.stopReplicas(stopReplicaRequest)
  val stopReplicaResponse = new StopReplicaResponse(
    stopReplicaRequest.correlationId,
```

```
        response.toMap,
        error)
      requestChannel.sendResponse(new Response(
        request,
        new BoundedByteBufferSend(stopReplicaResponse)))
      /* 此时需要检查 ReplicaManager 内部的不同 ReplicaFetcher 线程负责的 TopicAndPartition
        个数是否缩减为 0，如果缩减为 0，则可以关闭该 ReplicaFetcher 线程 */
      replicaManager.replicaFetcherManager.shutdownIdleFetcherThreads()
  }
```

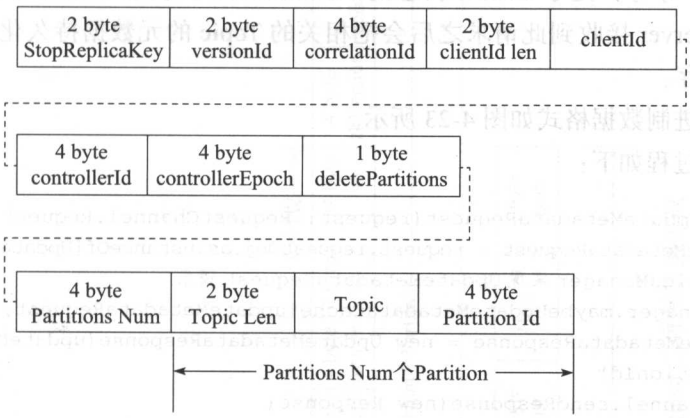


图 4-22　StopReplicaRequest 二进制数据组成

ReplicaManager 的 stopReplicas 流程在 4.3.2.2 小节已经详细描述，读者可以参考此小节，本节只是粗略描述下。StopReplicaRequest 请求中的 deletePartitions 参数决定是否删除该分区，当 Broker Server 接收到此请求之后，除了会暂停 ReplicaFetcher 线程拉取对应 TopicAndPartition 的数据之后，还会根据 deletePartitions 参数决定是否删除对应 TopicAndPartition 在当前 Broker Server 的数据，如果 deletePartitions 设置为 True，则会把对应 TopicAndPartition 的数据在当前 Broker Server 上清理掉。

当暂停了若干个 TopicAndPartition 的数据拉取之后，很有可能 ReplicaManager 内部的某些 ReplicaFetcher 线程空闲下来，没有任何拉取任务，此时应该尽早释放线程资源，防止资源占用，其具体实现过程如下：

```
def shutdownIdleFetcherThreads() {
  mapLock synchronized {
    val keysToBeRemoved = new mutable.HashSet[BrokerAndFetcherId]
    for ((key, fetcher) <- fetcherThreadMap) {
      /* 统计该 ReplicaFetcher 线程负责的 TopicAndPartition 个数是否缩减为 0 以下，如果
        是，则关闭该 ReplicaFetcher 线程 */
      if (fetcher.partitionCount <= 0) {
        fetcher.shutdown()
        keysToBeRemoved += key
      }
```

```
        }
        // 彻底清除掉被删除的 ReplicaFetcher 线程信息
        fetcherThreadMap --= keysToBeRemoved
      }
    }
```

4.3.5.7 UpdateMetadataRequest

当 Topic 的元数据发生改变时，例如 TopicAndPartition 的 Leader Replica，In-Sync Replicas，Assigned Replicas 等等，处于 Leader 状态的 KafkaController 会发送此请求至相关的 Broker Server，Broker Server 接收到此请求之后会把相关的 Topic 的元数据持久化至内存，并且仅仅是持久化至内存。

该请求的二进制数据格式如图 4-23 所示。

其详细实现过程如下：

```
def handleUpdateMetadataRequest(request: RequestChannel.Request) {
  val updateMetadataRequest = request.requestObj.asInstanceOf[UpdateMetadataRequest]
  // 通过 ReplicaManager 实现 UpdateMetadataRequest 请求
  replicaManager.maybeUpdateMetadataCache(updateMetadataRequest, metadataCache)
  val updateMetadataResponse = new UpdateMetadataResponse(updateMetadataRequest.
    correlationId)
  requestChannel.sendResponse(new Response(
    request,
    new BoundedByteBufferSend(updateMetadataResponse)))
  }
```

其中 metadataCache 保存了每个 TopicAndPartition 的状态，其组成如下：

```
private[server] class MetadataCache {
  // Map[Topic, mutable.Map[ 分区索引，分区状态 ]]
  private val cache: mutable.Map[String, mutable.Map[Int, PartitionStateInfo]] =
    new mutable.HashMap[String, mutable.Map[Int, PartitionStateInfo]]()
  // [Broker ID, Broker ID +IP 地址 + 端口号 ]
  private var aliveBrokers: Map[Int, Broker] = Map()
  ......
}
```

在利用 ReplicaManager 实现更新元数据的过程中调用的是 maybeUpdateMetadataCache 函数，那么为什么函数前缀为 maybe 呢？即为什么是可能更新呢？由于 Kafka 集群是分布式的，Broker Server 之间相互通信会存在时延情况，如果 Topic 的元数据连续变化两次，则 Leader 状态的 KafkaController 会连续下发两次更新请求，假设先发送的请求后达到目的端 Broker Server，后发送的请求先到达目的端 Broker Server，则目的端 Broker Server 把后发送的请求携带的 Topic 元数据更新至 metadataCache 之后，再接收到先发送的请求，该如何处理呢？很显然此时不应该更新先发送的请求所携带的 Topic 元数据，即 maybeUpdateMetadataCache 的实现过程如下：

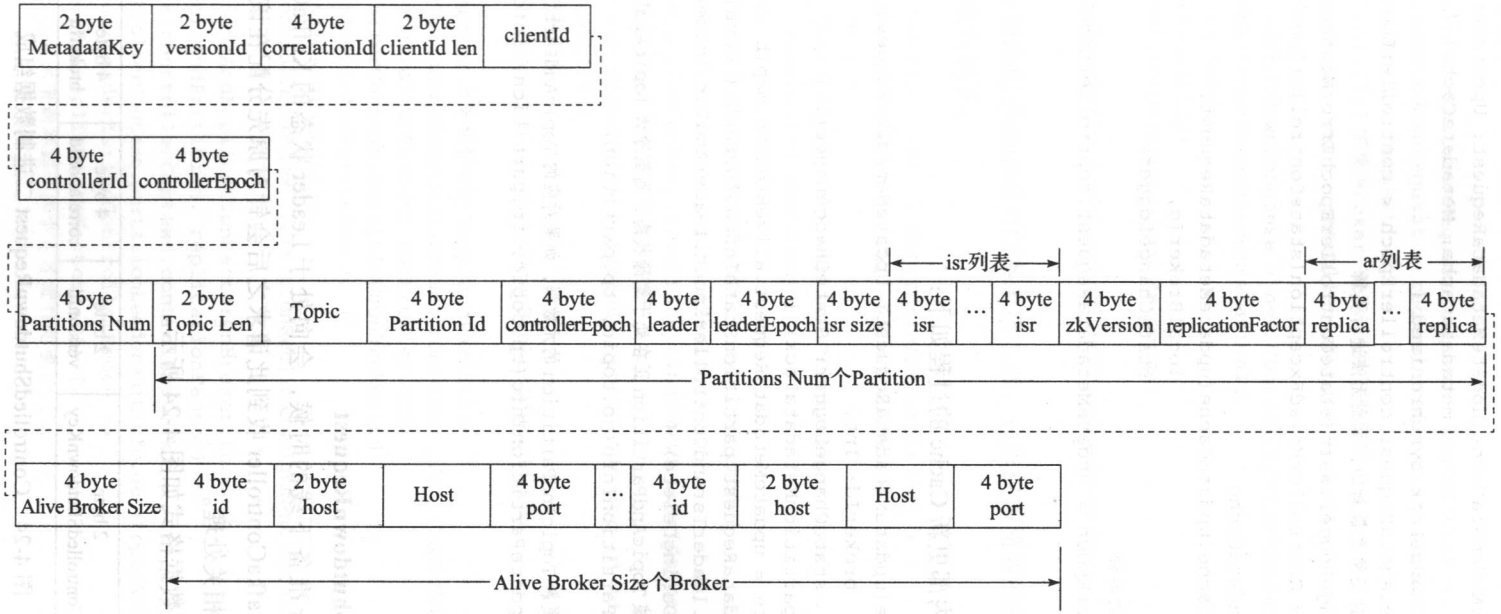


图 4-23　UpdateMetadataRequest 二进制数据组成

```
def maybeUpdateMetadataCache(updateMetadataRequest: UpdateMetadataRequest,
                             metadataCache: MetadataCache) {
  replicaStateChangeLock synchronized {
    if(updateMetadataRequest.controllerEpoch < controllerEpoch) {
      // 更新的时钟必须是最新的，否则无法进行更新
      stateChangeLogger.warn(stateControllerEpochErrorMessage)
      throw new ControllerMovedException(stateControllerEpochErrorMessage)
    } else {
      // 更新 metadataCache
      metadataCache.updateCache(updateMetadataRequest,
                                localBrokerId,
                                stateChangeLogger)
      // 更新当前的时钟
      controllerEpoch = updateMetadataRequest.controllerEpoch
    }
  }
}
```

MetadataCache 内部更新 Cache 的过程如下：

```
def updateCache(updateMetadataRequest: UpdateMetadataRequest,
                brokerId: Int,
                stateChangeLogger: StateChangeLogger) {
  inWriteLock(partitionMetadataLock) {
    aliveBrokers = updateMetadataRequest.aliveBrokers.map(b => (b.id, b)).toMap
    updateMetadataRequest.partitionStateInfos.foreach { case(tp, info) =>
      if (info.leaderIsrAndControllerEpoch.leaderAndIsr.leader == LeaderAndIsr.
        LeaderDuringDelete) {
        /* 如果该 TopicAndPartition 正在处于删除状态，则删除该 TopicAndPartition 的元数据 */
        removePartitionInfo(tp.topic, tp.partition)
      } else {
        /* 否则更新 TopicAndPartition 的元数据，如果对应的 TopicAndPartition 不存在，则创建 */
        addOrUpdatePartitionInfo(tp.topic, tp.partition, info)
      }
    }
  }
}
```

4.3.5.8 ControlledShutdownRequest

当 Broker Server 准备下线的时候，会向处于 Leader 状态的 KafkaController 发送此请求，Leader 状态的 KafkaController 收到此请求之后会针对原先分配在该 Broker Server 上的 TopicAndPartition 做相关处理。

该请求的二进制数据格式如图 4-24 所示。

2 byte ControlledShutdownKey	2 byte versionId	4 byte correlationId	4 byte brokerId

图 4-24 ControlledShutdownRequest 二进制数据组成

其详细实现过程如下：

```
def handleControlledShutdownRequest(request: RequestChannel.Request) {
  val controlledShutdownRequest = request.requestObj.asInstanceOf
    [ControlledShutdownRequest]
  // 通过 KafkaController 实现 Broker Server 的下线逻辑
  val partitionsRemaining = controller.shutdownBroker(controlledShutdownRequest.
    brokerId)
  val controlledShutdownResponse = new ControlledShutdownResponse(
    controlledShutdownRequest.correlationId,
    ErrorMapping.NoError,
    partitionsRemaining)
  requestChannel.sendResponse(new Response(
    request,
    new BoundedByteBufferSend(controlledShutdownResponse)))
}
```

KafkaController 模块的详细说明可以参考第 5 章，这里仅做简要的描述，KafkaController 的 shutdownBroker 流程如下：

```
def shutdownBroker(id: Int) : Set[TopicAndPartition] = {
  // 当前 KafkaController 必须为 Leader，否则不能处理该请求
  if (!isActive()) {
    throw new ControllerMovedException("Controller moved to another broker.
      Aborting controlled shutdown")
  }
  controllerContext.brokerShutdownLock synchronized {
    inLock(controllerContext.controllerLock) {
      if (!controllerContext.liveOrShuttingDownBrokerIds.contains(id))
        // 如果该 Broker 不存在，则抛出异常
        throw new BrokerNotAvailableException("Broker id %d does not exist."
          .format(id))
      // 添加进关机列表
      controllerContext.shuttingDownBrokerIds.add(id)
    }
    // 查找出该 Broker 上受到影响的 TopicAndPartition
    val allPartitionsAndReplicationFactorOnBroker: Set[(TopicAndPartition, Int)] =
      inLock(controllerContext.controllerLock) {
        controllerContext.partitionsOnBroker(id)
          .map(topicAndPartition => (
            topicAndPartition,
            controllerContext.partitionReplicaAssignment(topicAndPartition).size))
      }
    allPartitionsAndReplicationFactorOnBroker.foreach {
      case(topicAndPartition, replicationFactor) =>
        inLock(controllerContext.controllerLock) {
          controllerContext.partitionLeadershipInfo.get(topicAndPartition).foreach {
            currLeaderIsrAndControllerEpoch =>
              if (replicationFactor > 1) {
                // 只有副本因子大于 1 才进行处理
```

```
                if (currLeaderIsrAndControllerEpoch.leaderAndIsr.leader == id) {
                  /* 下线的 Broker Server 是某个 TopicAndPartition 的 leader，则利用
                     PartitionStateMachine 切换 Partition 状态 */
                  partitionStateMachine.handleStateChanges(
                    Set(topicAndPartition),
                    OnlinePartition,
                    controlledShutdownPartitionLeaderSelector)
                } else {
                  /* 下线的 Broker Server 是某个 TopicAndPartition 的 Assigned
                     Replicas 中的一个，则需要切换 Replica 的状态 *
                  brokerRequestBatch.newBatch()
                  // 向下线的 Broker Server 下发 StopReplicaRequest 请求
                  brokerRequestBatch.addStopReplicaRequestForBrokers(
                    Seq(id),
                    topicAndPartition.topic,
                    topicAndPartition.partition,
                    deletePartition = false)
                  brokerRequestBatch.sendRequestsToBrokers(
                    epoch,
                    controllerContext.correlationId.getAndIncrement)
                  // 利用 ReplicaStateMachine 切换副本状态
                  replicaStateMachine.handleStateChanges(    Set(PartitionAndReplica(
                      topicAndPartition.topic,
                      topicAndPartition.partition,
                      id)),
                    OfflineReplica)
                }
              }
            }
          }
        }
        def replicatedPartitionsBrokerLeads() = inLock(controllerContext.controllerLock) {
          controllerContext.partitionLeadershipInfo.filter {
            case (topicAndPartition, leaderIsrAndControllerEpoch) =>
              leaderIsrAndControllerEpoch.leaderAndIsr.leader == id && controllerContext.
                partitionReplicaAssignment(topicAndPartition).size > 1
          }.map(_._1)
        }
        /* 返回无法切换 Leader Replica 的 TopicAndPartition，这些无法切换 Leader Replica 的
           TopicAndPartition 位于准备下线的 Broker Server 上 */
        replicatedPartitionsBrokerLeads().toSet
      }
    }
```

在上面流程中主要区分为两种情况：

- 当下线的 Broker Server 为某些 TopicAndPartition 的 Leader 时，此时需要重新选举 TopicAndPartition 的 Leader，并且向相关的 Broker Server 发送 LeaderAndIsrRequest 请求通知其变化。PartitionStateMachine 的 Partition 状态切换在 5.3.2 小节会详细描述。
- 当下线的 Broker Server 位于某些 TopicAndPartition 的 Assigned Replicas 列表中时，

此时仅仅需要向相关的 Broker Server 发送 LeaderAndIsrRequest 请求通知其变化。ReplicaStateMachine 的 Replica 状态切换在 5.5.2 小节会详细描述。

4.3.5.9　OffsetCommitRequest

当高级消费者间隔一定时间将不同 Consumer Group 的消费情况提交至 Kafka 集群时会发送此请求，Broker Server 收到此请求之后会把详细的偏移量信息保存起来。

该请求的二进制数据格式如图 4-25 所示。

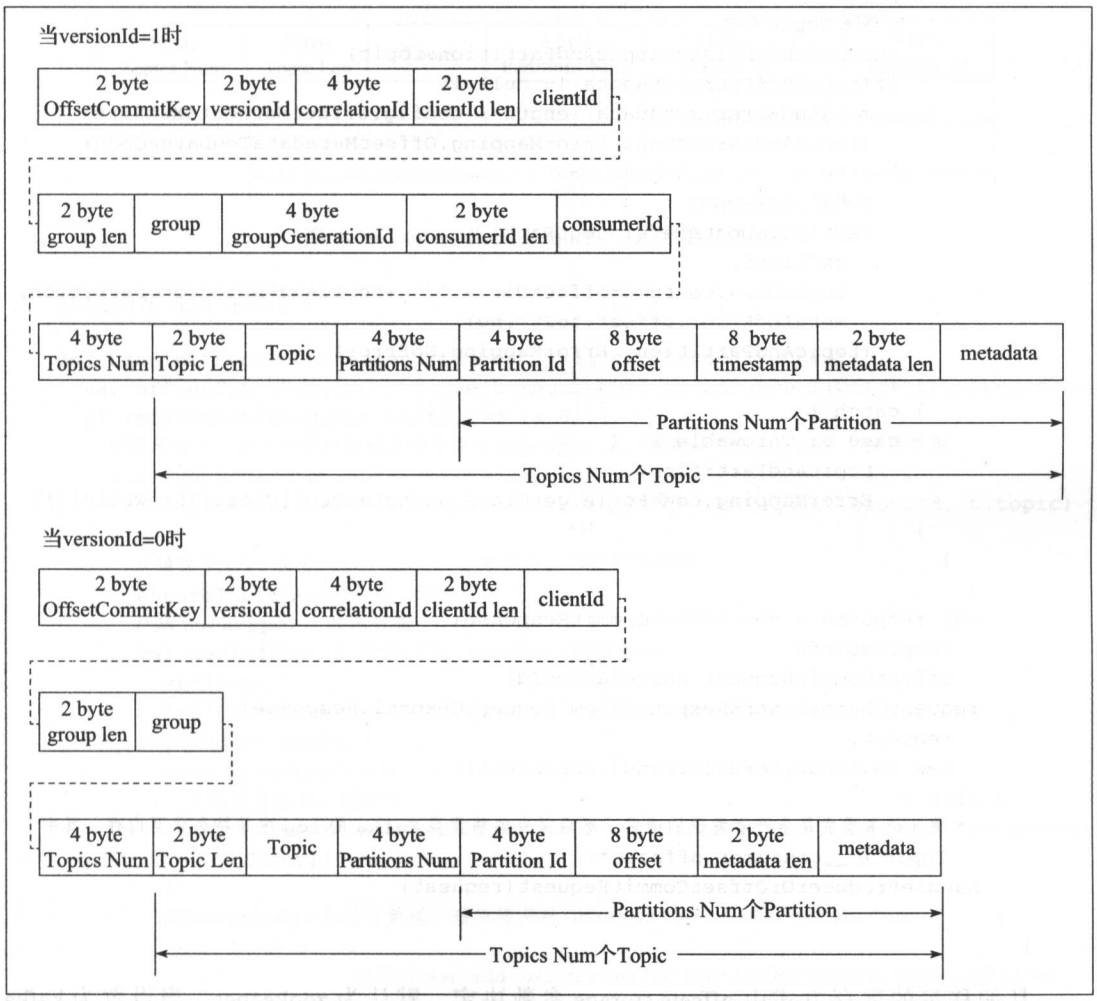


图 4-25　OffsetCommitRequest 二进制数据组成

其详细实现过程如下：

```
def handleOffsetCommitRequest(request: RequestChannel.Request) {
  val offsetCommitRequest = request.requestObj.asInstanceOf[OffsetCommitRequest]
```

```
if (offsetCommitRequest.versionId == 0) {
  // 版本为 0，将偏移量保存至 ZK
  val responseInfo = offsetCommitRequest.requestInfo.map{
    // metaAndError 内部包含了 offset
    case (topicAndPartition, metaAndError) => {
      // 组装不同 Consumer Group 在 Zookeeper 中的路径
      val topicDirs = new ZKGroupTopicDirs(
        offsetCommitRequest.groupId,
        topicAndPartition.topic)
      try {
        // 确保 Topic 存在
        ensureTopicExists(topicAndPartition.topic)
        if(metaAndError.metadata != null &&
          metaAndError.metadata.length > config.offsetMetadataMaxSize) {
          (topicAndPartition, ErrorMapping.OffsetMetadataTooLargeCode)
        } else {
          // 更新 Zookeeper
          ZkUtils.updatePersistentPath(
            zkClient,
            topicDirs.consumerOffsetDir + "/" +topicAndPartition.partition,
            metaAndError.offset.toString)
          (topicAndPartition, ErrorMapping.NoError)
        }
      } catch {
        case e: Throwable => (
          topicAndPartition,
          ErrorMapping.codeFor(e.getClass.asInstanceOf[Class[Throwable]]))
      }
    }
  }
  val response = new OffsetCommitResponse(
    responseInfo,
    offsetCommitRequest.correlationId)
  requestChannel.sendResponse(new RequestChannel.Response(
    request,
    new BoundedByteBufferSend(response)))
} else {
  /* 将此请求看成是生产者发送的请求，然后保存偏移量至 Kafka 的 log 中并持久化至内存，其中
    Topic 为 __consumer_offsets*/
  handleProducerOrOffsetCommitRequest(request)
}
}
```

其偏移量的保存方式由 offsets.storage 参数决定，默认为 zookeeper，当设置为 kafka 时，则将其保存在 Kafka 的 log 中并且持久化至内存，支持快速读取。

4.3.5.10 OffsetFetchRequest

当高级消费者想要查询不同 Consumer Group 的消费情况时会发送此请求，Broker Server 收到此请求之后会把详细的偏移量信息返回回来。

该请求的二进制数据格式如图 4-26 所示。

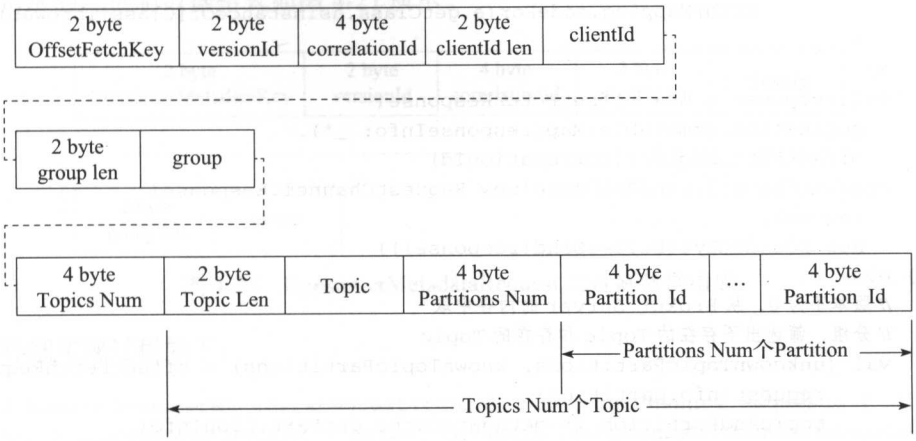


图 4-26　OffsetFetchRequest 二进制数据组成

其详细实现过程如下：

```
def handleOffsetFetchRequest(request: RequestChannel.Request) {
  val offsetFetchRequest = request.requestObj.asInstanceOf[OffsetFetchRequest]
  if (offsetFetchRequest.versionId == 0) {
    //版本为 0，由于之前的偏移量保存在 Zookeeper 上，因此从 Zookeeper 上拉取数据
    val responseInfo = offsetFetchRequest.requestInfo.map( t => {
    val topicDirs = new ZKGroupTopicDirs(offsetFetchRequest.groupId, t.topic)
    try {
      //确保 Topic 在 Broker Server 中存在，否则抛出异常
      ensureTopicExists(t.topic)
      //从 Zookeeper 上读取数据
      val payloadOpt = ZkUtils.readDataMaybeNull(
        zkClient,
        topicDirs.consumerOffsetDir + "/" + t.partition)._1
      payloadOpt match {
        case Some(payload) => {
          //如果有数据，则返回
          (t, OffsetMetadataAndError(offset=payload.toLong, error=ErrorMapping.
            NoError))
        }
        //Zookeepert 上没有数据，返回错误码 UnknownTopicOrPartitionCode
        case None => (t,
                      OffsetMetadataAndError(OffsetAndMetadata.InvalidOffset,
                                             OffsetAndMetadata.NoMetadata,
                                             ErrorMapping.
                                             UnknownTopicOrPartitionCode))
      }
    } catch {
      case e: Throwable =>
        (t, OffsetMetadataAndError(
```

```
                OffsetAndMetadata.InvalidOffset,
                OffsetAndMetadata.NoMetadata,
                ErrorMapping.codeFor(e.getClass.asInstanceOf[Class[Throwable]])))
        }
      })
      val response = new OffsetFetchResponse(
        collection.immutable.Map(responseInfo: _*),
        offsetFetchRequest.correlationId)
      requestChannel.sendResponse(new RequestChannel.Response(
        request,
        new BoundedByteBufferSend(response)))
    } else {
      // 版本不为 0，从 Broker Server 的内存中取
      // 分组，筛选出不存在的 Topic 和存在的 Topic
      val (unknownTopicPartitions, knownTopicPartitions) = offsetFetchRequest.
          requestInfo.partition(
          topicAndPartition => metadataCache.getPartitionInfo(
            topicAndPartition.topic,
            topicAndPartition.partition).isEmpty
      )
      // 针对不存在的 Topic，返回错误码 UnknownTopicOrPartitionCode
      val unknownStatus = unknownTopicPartitions.map(
        topicAndPartition => (topicAndPartition,
                              OffsetMetadataAndError.UnknownTopicOrPartition)).toMap
      // 针对存在的 Topic，从 OffsetManager 的内存中取
      val knownStatus =
        if (knownTopicPartitions.size > 0)
          offsetManager.getOffsets(offsetFetchRequest.groupId, knownTopicPartitions).
          toMap
        else
          Map.empty[TopicAndPartition, OffsetMetadataAndError]
      val status = unknownStatus ++ knownStatus
      val response = OffsetFetchResponse(status, offsetFetchRequest.correlationId)
      requestChannel.sendResponse(new RequestChannel.Response(
        request,
        new BoundedByteBufferSend(response)))
    }
  }
```

OffsetCommitRequest 和 OffsetFetchRequest 组成一对关于偏移量保存和读取的请求，两者都根据 offsets.storage 的配置决定偏移量的写入和读取路径，考虑到 Zookeeper 不适合频繁的写入场景，因此建议用户配置 offsets.storage 为 Kafka。

4.3.5.11 ConsumerMetadataRequest

当 offsets.storage 配置为 kafka 时，其不同 Consumer Group 的偏移量都是保存在 Topic 为“__consumer_offsets”的日志里面的，每个具体的 Consumer Group 的偏移量保存在特定的分区里面，当客户端想要知道分配给某个具体的 Consumer Group 所在的特定分区状态时就会发送此请求，Broker Server 收到此请求之后会把特定 Consumer Group 所在的分区的

Leader Replica 索引返回回来。

该请求的二进制数据格式如图 4-27 所示。

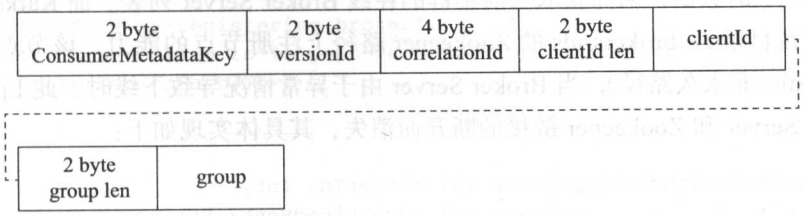


图 4-27　ConsumerMetadataRequest 二进制数据组成

其详细实现过程如下：

```
def handleConsumerMetadataRequest(request: RequestChannel.Request) {
  val consumerMetadataRequest = request.requestObj.asInstanceOf[ConsumerMetadataRequest]
  //获取保存该 Consumer Group 偏移量的分区索引
  val partition = offsetManager.partitionFor(consumerMetadataRequest.group)
  /*获取 Topic 为“__consumer_offsets”的元数据，包括 Topic 名字，分区的 Leader Replica,
    Assigned Replicas, In-Sync Replicas*/
  val offsetsTopicMetadata = getTopicMetadata(Set(OffsetManager.OffsetsTopicName)).
    head
  val errorResponse = ConsumerMetadataResponse(
    None,
    ErrorMapping.ConsumerCoordinatorNotAvailableCode,
    consumerMetadataRequest.correlationId)
  val response =
    /*如果对于分区其 leader 存在，则返回其索引，否则返回错误码
      ConsumerCoordinatorNotAvailableCode*/
    offsetsTopicMetadata.partitionsMetadata.find(_.partitionId == partition).
      map { partitionMetadata =>
        partitionMetadata.leader.map { leader =>
          ConsumerMetadataResponse(Some(leader),
          ErrorMapping.NoError,
          consumerMetadataRequest.correlationId)
      }.getOrElse(errorResponse)
    }.getOrElse(errorResponse)
  requestChannel.sendResponse(new RequestChannel.Response(
    request,
    new BoundedByteBufferSend(response)))
}
```

4.4　KafkaHealthcheck

KafkaHealthcheck 主要提供 Broker Server 健康状态的上报。Broker Server 的健康状态本质上就是指 Broker Server 是否在线，如果 Broker Server 在线，则说明当前处于健康状态；如果 Broker Server 离线，则说明当前处于死亡状态。

那么 Broker Server 是如何上报健康状态呢？在 5.6.6 小节将要提到 BrokerChangeListener 监听器，它通过监听目录为 /brokers/ids 的 Zookeeper 路径，当发现有数据发生变化时，则获取当前目录下的数据，从而获取当前集群的在线 Broker Server 列表。而 KafkaHealthcheck 正是提供了在目录为 /brokers/ids 的 Zookeeper 路径上注册节点的能力，该节点所在路径为 EphemeralPath（非永久路径），当 Broker Server 由于异常情况导致下线时，此 EphemeralPath 随着 Broker Server 和 Zookeeper 链接的断开而消失，其具体实现如下：

```
class KafkaHealthcheck(private val brokerId: Int,
                       private val advertisedHost: String,
                       private val advertisedPort: Int,
                       private val zkSessionTimeoutMs: Int,
                       private val zkClient: ZkClient) extends Logging {
  val sessionExpireListener = new SessionExpireListener
  // KafkaHealthcheck 启动的时候注册节点
  def startup() {
    // 订阅链接的状态变化
    zkClient.subscribeStateChanges(sessionExpireListener)
    // 注册节点
    register()
  }
  def register() {
    // 获取主机名
    val advertisedHostName =
      if(advertisedHost == null || advertisedHost.trim.isEmpty)
        InetAddress.getLocalHost.getCanonicalHostName
      else
        advertisedHost
    // 获取 jmx 端口号
    val jmxPort = System.getProperty("com.sun.management.jmxremote.port", "-1").toInt
    /* 通过 zkClient 的 createEphemeral 方法创建瞬时节点，该节点的数据为 BrokerId，主机
       名，客户端链接端口，JMX 端口 */
    ZkUtils.registerBrokerInZk(
      zkClient,
      brokerId,
      advertisedHostName,
      advertisedPort,
      zkSessionTimeoutMs,
      jmxPort)
  }
}
```

那么当 zkClient 发生重连的时候，之前创建的 EphemeralPath（非永久路径）会消失，此时如果 SessionExpireListener 监测到新连接产生时就应该重新在之前的节点上注册该节点数据，其具体实现如下：

```
class SessionExpireListener() extends IZkStateListener {
  // 之前的 Zookeeper 会话超时，新的的 Zookeeper 会话刚建立时触发
  def handleNewSession() {
```

```
        info("re-registering broker info in ZK for broker " + brokerId)
        // 通过 register 重新在 /brokers/ids 上注册该节点数据
        register()
        info("done re-registering broker")
        info("Subscribing to %s path to watch for new topics".format(ZkUtils.
          BrokerTopicsPath))
      }
    }
  }
```

4.5 TopicConfigManager

Kafka 提供对 Topic 配置参数的在线修改能力，修改完成之后无需重新启动 Kafka 集群，在线生效。Topic 配置参数包括：数据文件的大小、索引文件的大小、索引项的粒度、日志文件保留的策略等等。

Topic 的配置参数位于路径为 /config/topics/[topic] 的 Zookeeper 上，Broker Server 内部为了避免针对每个 Topic 都在相关路径上建立监听器，对外提供了一个被通知的路径，其位于 /brokers/config_changes，如果监测到该路径上发生变化，则读取该路径上的数据，获取配置文件待更新的 Topic，然后再从 /config/topics/[topic] 上加载最新的配置文件。为了防止重复更新或者错误更新，Broker Server 规定了在 /brokers/config_changes 目录上创建节点的格式，其节点名为 config_changes_x，其中 x 代表更新的时序，每次需要递增，Broker Server 只更新当前最新的通知，不更新以前的通知。为了防止该 /brokers/config_changes 路径下节点数不断增多，Broker Server 提供了超时删除机制，当超过一定的时间就删除该通知。

因此客户端更新 Topic 配置文件的步骤如下：

1）在 /config/topics/[topic] 目录输入 Topic 的配置参数。

2）在 /brokers/config_changes 目录通知每个 Broker Server 更新内部的 log 配置参数。

TopicConfigManager 提供监听 /brokers/config_changes 目录下的通知，然后从 /config/topics/[topic] 加载配置的功能。它在启动阶段会在 /brokers/config_changes 目录下建立监听器，同时处理当前存在的通知，其具体实现如下：

```
class TopicConfigManager(private val zkClient: ZkClient,
                         private val logManager: LogManager,
                         private val changeExpirationMs: Long = 15*60*1000,
                         private val time: Time = SystemTime) extends Logging {
  def startup() {
    // 确保 /config/changes 路径存在，如果不存在，则创建
    ZkUtils.makeSurePersistentPathExists(zkClient, ZkUtils.TopicConfigChangesPath)
    // 在 /config/changes 目录上建立 ConfigChangeListener 监听器
    zkClient.subscribeChildChanges(ZkUtils.TopicConfigChangesPath, ConfigChangeListener)
    // 处理 /config/changes 上遗留的通知
    processAllConfigChanges()
  }
}
```

processAllConfigChanges 将从 /config/changes 目录上读取被通知的具体内容，然后根据时序，只更新大于当前更新的序号，同时删除过时的通知，其具体实现如下：

```
//notifications 为 /config/changes 下的节点名
private def processConfigChanges(notifications: Seq[String]) {
  if (notifications.size > 0) {
    //记录当前时间
    val now = time.milliseconds
    //获取 TopicAndPartition 和 Log 的映射
    val logs = logManager.logsByTopicPartition.toBuffer
    //将 Log 按照 Topic 分组
    val logsByTopic = logs.groupBy(_._1.topic).mapValues(_.map(_._2))
    for (notification <- notifications) {
      //获取时序 id
      val changeId = changeNumber(notification)
      //只有大于当前时序 id 才生效
      if (changeId > lastExecutedChange) {
        val changeZnode = ZkUtils.TopicConfigChangesPath + "/" + notification
        //获取该 /config/changes/config_change_x 节点的数据
        val (jsonOpt, stat) = ZkUtils.readDataMaybeNull(zkClient, changeZnode)
        if(jsonOpt.isDefined) {
          val json = jsonOpt.get
          //提取其中的 topic
          val topic = json.substring(1, json.length - 1)
          //topic 存在
          if (logsByTopic.contains(topic)) {
            //先加载当前默认的配置参数
            val props = new Properties(logManager.defaultConfig.toProps)
            //然后从 /config/topics/[topic] 目录加载最新的参数，会覆盖旧的参数
            props.putAll(AdminUtils.fetchTopicConfig(zkClient, topic))
            //将 Properties 转化为 LogConfig
            val logConfig = LogConfig.fromProps(props)
            //更新每个 Log 的配置参数
            for (log <- logsByTopic(topic))
              log.config = logConfig
            //删除过时的通知
            purgeObsoleteNotifications(now, notifications)
          }
        }
        //更新时序 id
        lastExecutedChange = changeId
      }
    }
  }
}
```

在删除过时的通知中，TopicConfigmanager 会比对当前时间 now 和 Zookeeper 上节点创建的时间差，如果大于 changeExpirationMs，即 15 × 60 × 1000 毫秒，则会把相应的通知删除掉，其具体实现如下：

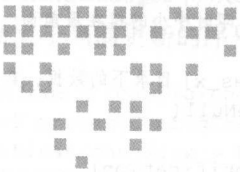
```
private def purgeObsoleteNotifications(now: Long, notifications: Seq[String]) {
  // 将 /brokers/config_changes 目录上的节点升序排序，即时序小的排在最前面
  for(notification <- notifications.sorted) {
    // 读取 /brokers/config_changes/[config_changes_x] 目录下的数据，数据存在才有效
    val (jsonOpt, stat) = ZkUtils.readDataMaybeNull(
      zkClient,
      ZkUtils.TopicConfigChangesPath + "/" + notification)
    if(jsonOpt.isDefined) {
      val changeZnode = ZkUtils.TopicConfigChangesPath + "/" + notification
      /* 判断当前时间和节点创建时间间隔是否大于 changeExpirationMs，如果超过的话，则删除该节点 */
      if (now - stat.getCtime > changeExpirationMs) {
        ZkUtils.deletePath(zkClient, changeZnode)
      } else {
        return
      }
    }
  }
}
```

因此当 ConfigChangeListener 监听器监测到 /config/changes 目录上数据发生变化时，只会更新最新的通知，系统启动阶段已经更新过的通知则不会处理，如下所示：

```
object ConfigChangeListener extends IZkChildListener {
  override def handleChildChange(path: String, chillins: java.util.List[String]) {
    try {
      import JavaConversions._
      /* 此时 lastExecutedChange 是最新的，Broker Server 只会更新大于 lastExecutedChange
         的通知 */
      processConfigChanges(chillins: mutable.Buffer[String])
    } catch {
      case e: Exception => error("Error processing config change:", e)
    }
  }
}
```

4.6　本章小结

本章详细描述了 Broker 九大基本模块内部的实现原理，其中 KafkaApis 真正负责具体的业务逻辑实现，它需要利用 LogManager、ReplicaManager、OffsetManager 和 KafkaHealthcheck 四大模块来帮助完成具体的业务实现。在 KafkaApis 模块中，Broker 对外暴露了十一条通信协议，这十一通信协议是理解整个 Broker 对外提供服务的关键。



Chapter 5 第 5 章

Broker 的控制管理模块

在 3.2 节中提到 KafkaController 是 Kafka 集群的控制管理模块，虽然每个 Broker 内部都会存在一个 KafkaController 模块，但是有且只有一个 Broker 内部的 KafkaController 模块对外提供控制管理 Kafka 集群的功能，例如负责 Topic 的创建、分区的重分配以及分区副本 Leader 的重新选举等等。本章将详细描述 KafkaController 内部的实现原理，包括 KafkaController 之间的选举策略、KafkaController 的初始化、KafkaController 维护分区状态和副本状态的机制、KafkaController 内部的监听器、KafkaController 提供的负载均衡机制、KafkaController 提供的 Topic 删除机制以及 KafkaController 之间的通信原理。读者理解了 KafkaController 模块，也就理解了整个 Kafka 集群内部的管理机制。

5.1 KafkaController 的选举策略

在 Kafka 集群中，每一个 Broker Server 内部都会启动各自的 KafkaController 模块，但是只有其中一个 KafkaController 会成为 Leader 状态，其他都是 Follower 状态。KafkaController 模块的 Leader 和 Follower 的选举是基于 Zookeeper 实现的，而 Zookeeper 本身是一个分布式应用程序协调服务。KafkaController 模块运行起来之后尝试在 Zookeeper 的相同路径上创建瞬时节点（Ephemeral Node），有且只有一个 KafkaController 会创建成功，其他都会创建失败。那么何为瞬时节点呢？即读写 Zookeeper 模块的客户端会维护和 Zookeeper 集群的连接，当该连接断开的时候，通过此连接之前创建的瞬时节点都会消失，所以说该节点是瞬时的，不是永久存在的。接着当 Leader 状态的 KafkaController 离线的时候，其内部的 Zookeeper 客户端就会失去和 Zookeeper 集群的连接，那么此时其他 KafkaController 观察到

数据节点消失之后，就会重新尝试创建节点。多个 Zookeeper 客户端在 Zookeeper 集群的相同路径上创建瞬时节点时的原子性由 Zookeeper 保证，即有且只有一个 Zookeeper 客户端会创建成功，其大致流程如图 5-1 所示。

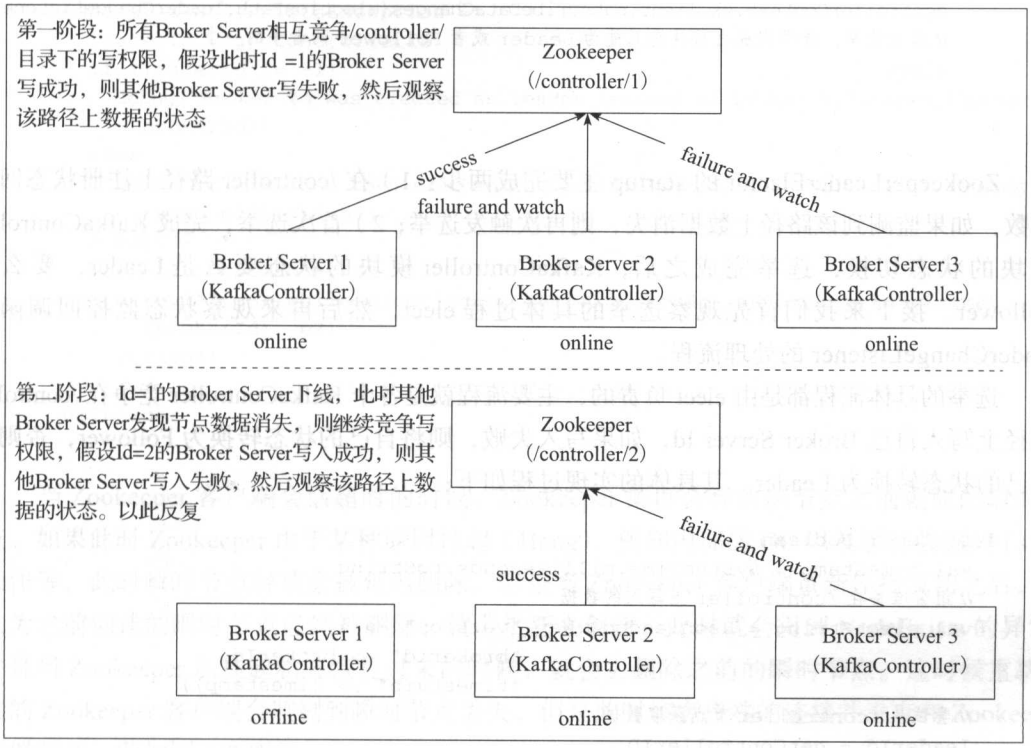


图 5-1　KafkaController 的选举

KafkaController 模块中负责状态管理的类为 ZookeeperLeaderElector，从字面意思上就可以看出是基于 Zookeeper 的 Leader 选举器，其构造函数组成如下：

```
class ZookeeperLeaderElector(controllerContext: ControllerContext,
                             electionPath: String,
                             onBecomingLeader: () => Unit,
                             onResigningAsLeader: () => Unit,
                             brokerId: Int)
  extends LeaderElector with Logging {
  ......
}
```

其中 controllerContext 为 Controller 上下文，里面包含了当前 Topic 的元数据信息以及集群的元数据信息等；electionPath 为多个 KafkaController 竞争写的路径，其值为 /controller；onBecomingLeader 为状态转换成 Leader 时候的回调函数；onResigningAsLeader 为状态转换成 Follower 时候的回调函数；brokerId 为当前 Broker Server 的 Id。ZookeeperLeaderElector

一旦启动（startup）之后就会负责 KafkaController 模块的状态转换，其启动流程如下：

```
def startup {
  inLock(controllerContext.controllerLock) {
    //负责观察数据节点状态，当数据节点消失时可以触发再次选举
    controllerContext.zkClient.subscribeDataChanges(electionPath,leaderChangeListener)
    //首次选举，选举完成之后状态只能为 Leader 或者 Follower 两者中的一个
    elect
  }
}
```

ZookeeperLeaderElector 的 startup 主要完成两步：1）在 /controller 路径上注册状态回调函数，如果监测到该路径上数据消失，则再次触发选举；2）首次选举，完成 KafkaController 模块的状态切换，选举完成之后，KafkaController 模块的状态要么是 Leader，要么是 Follower。接下来我们首先观察选举的具体过程 elect，然后再来观察状态监控回调函数 leaderChangeListener 的处理流程。

选举的具体流程都是由 elect 负责的，主要流程就是各个 KafkaController 竞争在 /controller 路径上写入自己 Broker Server Id，如果写入失败，则将自己的状态转换为 Follower，否则将自己的状态转换为 Leader。其具体的实现过程如下：

```
def elect: Boolean = {
  val timestamp = SystemTime.milliseconds.toString
  //组装准备在 /controller 上写入的数据
  val electString = Json.encode(Map("version" -> 1,
                                    "brokerid" -> brokerId,
                                    "timestamp" -> timestamp))
  //尝试从 /controller 节点获取数据
  leaderId = getControllerID
  //如果获取到数据，则退出选举
  if(leaderId != -1) {
    debug("Broker %d has been elected as leader, so stopping the election format
      (leaderId))
    return amILeader
  }
  try {
  //尝试在 /controller 路径上写入数据
  createEphemeralPathExpectConflictHandleZKBug(
    controllerContext.zkClient,
    electionPath,
    electString,
    brokerId,
    (controllerString : String, leaderId : Any) =>
      KafkaController.parseControllerId(controllerString) == leaderId.
        asInstanceOf[Int],
    controllerContext.zkSessionTimeout)
    //写入成功，则状态转换为 Leader
    info(brokerId + " successfully elected as leader")
    leaderId = brokerId
```

```
        //触发对应的回调函数
        onBecomingLeader()
    } catch {
      /* 很不幸，竞争失败，被集群中剩余的某个 KafkaController 抢先获得写权限，写入失败，此时捕获
        到节点已存在的异常 */
      case e: ZkNodeExistsException =>
        leaderId = getControllerID
        if (leaderId != -1)
          debug("Broker %d was elected as leader instead of broker %d".format(leaderId,
            brokerId))
        else
          warn("A leader has been elected but just resigned, this will result inanother
            round ofelection")
        //其他异常
        case e2: Throwable =>
          error("Error while electing or becoming leader on broker %d".format
            (brokerId), e2)
          resign()
      }
      amILeader
    }
```

当 Zookeeper 客户端会话超时的时候，Zookeeper 集群会释放该节点之前创建的瞬时节点，如果此时 Zookeeper 由于某种原因挂起（Hang），例如内部的 GC 以及缓存数据的 Fsync 操作等，此时瞬时节点释放会被延迟删除。但是当 Zookeeper 客户端重新连接成功之后，会认为之前创建的瞬时节点已经被删除，就会重新去创建，此时就会得到 NodeExists 的异常。并且当 Zookeeper 从挂起状态恢复过来的时候，就会去删除之前的瞬时节点。这时候重新连接的 Zookeeper 客户端会监测到瞬时节点丢失，但是此时该客户端的连接并没有和 Zookeeper 集群断开，即如图 5-2 所示。

因此 ZookeeperLeaderElector 在处理这种情况时采取了一点小的技巧，规避了 Zookeeper 集群在释放瞬时节点时存在的问题，如下所示：

```
def createEphemeralPathExpectConflictHandleZKBug(
  zkClient: ZkClient,
  path: String,
  data: String,
  expectedCallerData: Any,
  checker: (String, Any) => Boolean,
  backoffTime: Int): Unit = {
  while (true) {
    try {
      //尝试写入数据
      createEphemeralPathExpectConflict(zkClient, path, data)
        return
    } catch {
      case e: ZkNodeExistsException => {
        //节点已经存在，则读取节点数据
        ZkUtils.readDataMaybeNull(zkClient, path)._1 match {
```

```
            case Some(writtenData) => {
              if (checker(writtenData, expectedCallerData)) {
                /* 实际数据和准备写入的数据一致，则说明是同一个客户端不同的旧连接的数据还没有被
                   释放，需要一直等待其被释放之后，才能再次尝试写入数据 */
                Thread.sleep(backoffTime)
              } else {
                /* 实际数据和准备写入的数据不一致，则说明已经被其他客户端抢占了写权限，抛出异常 */
                throw e
              }
            }
            case None =>
          }
        }
        case e2: Throwable => throw e2
      }
    }
  }
```

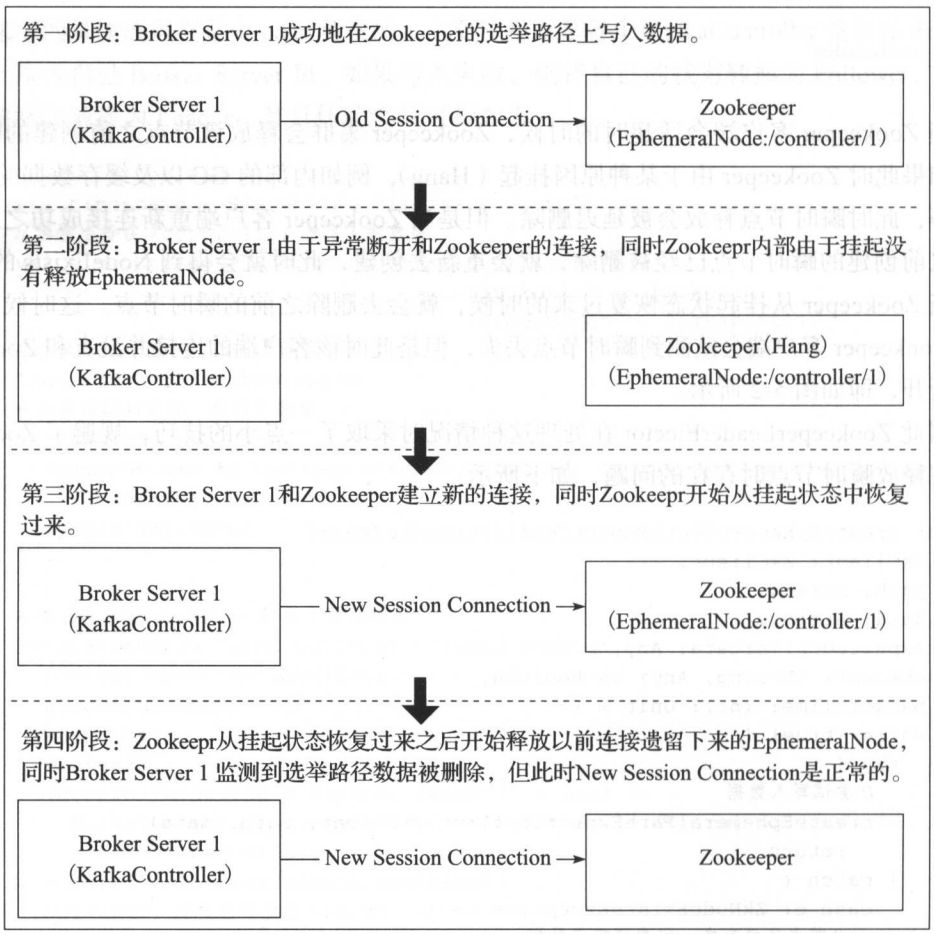


图 5-2 Zookeeper Bug

可见，所谓的技巧就是通过判断准备写入数据和实际数据的一致性，如果一致，则说明是由相同客户端的不同连接写入所导致的，如果不一致，则说明是由不同客户端写入所导致的。针对一致的情况，则必须等待 Zookeeper 集群释放之前连接创建的 EphemeralNode，然后再进行选举。

leaderChangeListener 负责 KafkaController 的状态变更，并监控 Zookeeper 选举路径上数据的状态，如果发现数据被删除，则会再次触发选举，其大致流程如下所示：

```
class LeaderChangeListener extends IZkDataListener with Logging {
    // 当选举路径上的数据被删除时触发
    def handleDataDeleted(dataPath: String) {
      inLock(controllerContext.controllerLock) {
        if(amILeader)
          // 如果之前自己是 Leader，则需要清理一些资源
          onResigningAsLeader()
        // 再次触发选举
        elect
      }
    }
  }
```

5.2　KafkaController 的初始化

KafkaController 模块的初始化是由 ZookeeperLeaderElector 决定的，当 KafkaController 被选举为 Leader 时会触发调用 onBecomingLeader 回调函数，当 KafkaController 被选举为 Follower 时会触发调用 onResigningAsLeader 回调函数，其中以上两个回调函数真正的实现分别为 ZookeeperLeaderElector 的 onControllerFailover 处理流程和 ZookeeperLeaderElector 的 onControllerResignation 处理流程。

5.2.1　Leader 状态下 KafkaController 的初始化

当 KafkaController 被选举为 Leader 时会触发调用 onBecomingLeader 回调函数，也就是 ZookeeperLeaderElector 的 onControllerFailover 处理函数，其具体实现如下：

```
def onControllerFailover() {
  if(isRunning) {
    info("Broker %d starting become controller state transition".format(config.brokerId))
    // 初始化集群内部时钟
    readControllerEpochFromZookeeper()
    incrementControllerEpoch(zkClient)
    // 注册各种监听函数
    registerReassignedPartitionsListener()
    registerPreferredReplicaElectionListener()
    partitionStateMachine.registerListeners()
    replicaStateMachine.registerListeners()
    // 初始化 Controller 上下文，即集群内部元数据信息
    initializeControllerContext()
```

```
        //初始化 Replica 状态，启动副本状态转化
        replicaStateMachine.startup()
        //初始化 Partition 状态，启动分区状态转化
        partitionStateMachine.startup()
        controllerContext.allTopics.foreach(topic =>
          partitionStateMachine.registerPartitionChangeListener(topic))
        info("Broker %d is ready to serve as the new controller with
          epoch %d".format(config.brokerId, epoch))
        //切换状态为 RunningAsController
        brokerState.newState(RunningAsController)
        //处理集群初始化之前用户下发的 PartitionReassignment 请求和 PreferredReplica 请求
        maybeTriggerPartitionReassignment()
        maybeTriggerPreferredReplicaElection()
        //同步集群元数据信息给其他 KafkaController
        sendUpdateMetadataRequest(controllerContext.liveOrShuttingDownBrokerIds.toSeq)
        if (config.autoLeaderRebalanceEnable) {
          //启动负载均衡线程
          info("starting the partition rebalance scheduler")
          autoRebalanceScheduler.startup()
          autoRebalanceScheduler.schedule("partition-rebalance-thread",
                                          checkAndTr iggerPartitionRebalance,
                                          5,
                                          config.leaderImbalanceCheckIntervalSeconds,
                                          TimeUnit.SECONDS)
        }
        //启动 Topic 删除线程
        deleteTopicManager.start()
      }
      else
        info("Controller has been shut down, aborting startup/failover")
    }
```

其大致上可以分为以下几步：

1）初始化 Kafka 集群内部的时钟，其时钟的具体数值是存放在 Zookeeper 上的，路径为 /controller_epoch，Broker Server 利用此值区分请求的时效性。

2）注册各种监听函数，由于 Kafka 把元数据持久化在 Zookeeper 上，因此 KafkaController 针对 Zookeeper 的不同目录注册不同的监听函数，监听函数的分布情况如图 5-3 所示。

其中在 /admin/reassign_partitions 目录上注册 PartitionReassignedListener，响应用户下发的重分配 Topic 分区的请求；在 /admin/preferred_replica_election 目录上注册 PreferredReplicaElectionListener，响应用户下发的重新选举 Topic 分区副本的请求；在 /admin/delete_topics 目录上注册 DeleteTopicsListener，响应用户下发的删除 Topic 的请求；在 /brokers/topics 目录上注册 TopicChangeListener，响应用户下发的创建 Topic 的请求；在 /brokers/topics/Topic-0（具体的某个 Topic）目录上注册 AddPartitionsListener，响应用户下发的增加 Topic 分区的请求；在 /brokers/topics/Topic-0（具体的某个 Topic）/0（分区索引）/state 目录上注册 ReassignedPartitionsIsrChangeListener，处理 Topic 分区的 ISR 列表的变化；在 /brokers/ids 目录上注册 BrokerChangeListener，处理 Broker 上下线所引起的 Topic 元数据信息的变化。

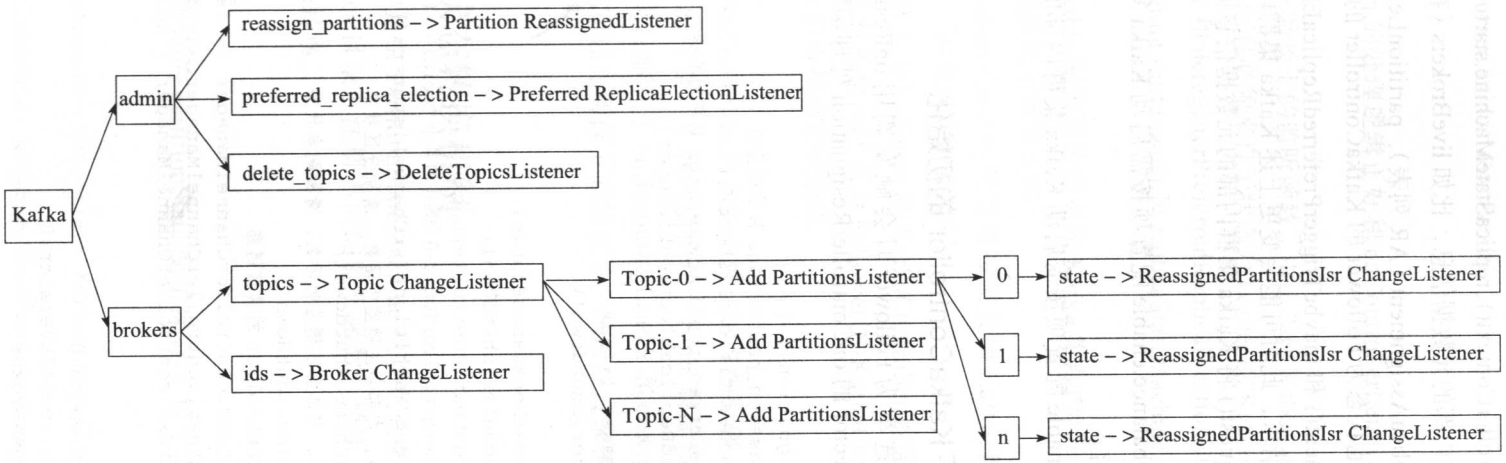


图 5-3　KafkaController 内监听函数的分布情况

3）通过 initializeControllerContext()、replicaStateMachine.startup() 和 partitionStateMachine.startup() 初始化 Kafka 集群内部的元数据信息，比如 liveBrokers（在线 Broker 列表）、allTopics（Topic 列表）、partitionReplicaAssignment（AR 列表）、partitionLeadershipInfo（ISR 列表）等等，以及建立和集群内其他状态为 Follower 的 KafkaController 的通信链路，同时通过 maybeTriggerPartitionReassignment() 和 maybeTriggerPreferredReplicaElection() 处理 Kafka 集群启动前没有及时处理的用户请求，此时可能会变更上述 Kafka 集群内部的元数据信息，最后通过 sendUpdateMetadataRequest() 将 Kafka 集群内部的元数据信息同步给其他状态为 Follower 的 KafkaController。

4）根据 auto.leader.rebalance.enable 配置项按需启动 Kafka 集群内部的负载均衡线程，其默认为 true，即默认开启。

5）根据 delete.topic.enable 配置项按需启动 Kafka 集群内部的 Topic 删除线程，其默认为 false，即默认关闭。

5.2.2 Standby 状态下 KafkaController 的初始化

当 KafkaController 被选举为 Follower 时会触发调用 onResigningAsLeader 回调函数，也就是 ZookeeperLeaderElector 的 onControllerResignation 处理函数，其具体实现如下：

```
def onControllerResignation() {
  // 取消针对 /admin/reassign_partitions 目录的监听
  deregisterReassignedPartitionsListener()
  // 取消针对 /admin/preferred_replica_election 目录的监听
  deregisterPreferredReplicaElectionListener()
  // 关闭 Topic 删除线程，
  if (deleteTopicManager != null)
    deleteTopicManager.shutdown()
  // 关闭负载均衡线程
  if (config.autoLeaderRebalanceEnable)
    autoRebalanceScheduler.shutdown()
  inLock(controllerContext.controllerLock) {
    /* 取消针对类似 /brokers/topics/Topic-0（具体的某个 Topic）/0（分区索引）/state 目录的监控 */
    deregisterReassignedPartitionsIsrChangeListeners()
    // 关闭分区状态转换机，内部会注销监听函数，清除分区状态
    partitionStateMachine.shutdown()
    // 关闭副本状态转换机，内部会注销监听函数，清除副本状态
    replicaStateMachine.shutdown()
    // 关闭和其他 KafkaController 的通信链路
    if(controllerContext.controllerChannelManager != null) {
       controllerContext.controllerChannelManager.shutdown()
       controllerContext.controllerChannelManager = null
    }
    // 重置集群内部时钟
    controllerContext.epoch=0
    controllerContext.epochZkVersion=0
    // 切换状态为 RunningAsBroker
```

```
        brokerState.newState(RunningAsBroker)
    }
}
```

onControllerResignation 的处理逻辑正好和 onControllerFailover 的处理逻辑相反，其大致可以分为以下几步：

1）取消各种 Zookeeper 路径上的监听函数，由于当前 KafkaController 被选举为 Follower，所以有关集群的所有元数据信息必须来自于状态为 Leader 的 KafkaController。因此 deregisterReassignedPartitionsListener 和 deregisterPreferredReplicaElectionListener 会取消部分监听函数，并且 partitionStateMachine.shutdown() 和 replicaStateMachine.shutdown() 也会取消部分监听函数。

2）根据 delete.topic.enable 配置项按需关闭 Kafka 集群内部的 Topic 删除线程，其默认为 false，即默认关闭，因此默认不需要关闭。

3）根据 auto.leader.rebalance.enable 配置项按需关闭 Kafka 集群内部的负载均衡线程，其默认为 true，即默认开启，因此默认需要关闭。

4）断开和集群内其他状态为 Follower 的 KafkaController 的通信链路，因为只有状态为 Leader 的 KafkaController 才有资格向状态为 Follower 的 KafkaController 发送请求，状态为 Follower 的 KafkaController 只有被动接收请求。

5）重置集群内部时钟。

接下来将主要讲解 KafkaController 内部各个模块的详细原理，即 PartitionStateMachine、ReplicaStateMachine、各种监听器、负载均衡模块、Topic 删除模块，以及 KafkaController 的通信链路模块。读者如果目前对于 onBecomingLeader 和 onResigningAsLeader 流程还是不熟悉的话，请不要慌张，可以先阅读下面的章节，然后反过来再看这个章节，这样会加深对相关知识点的理解。

5.3　Topic 的分区状态转换机制

Topic 的分区状态维护是由 PartitionStateMachine 模块负责的，该模块通过在 /brokers/topics 和 /admin/delete_topics 目录上注册不同的监听函数，监听 Topic 的创建和删除事件，从而触发 Topic 分区状态的转换。

5.3.1　分区状态的分类

PartitionStateMachine 内部的 partitionState 变量保存了每个具体的 Topic 的分区状态，如下所示：

```
class PartitionStateMachine(controller: KafkaController) extends Logging {
  ......
  private val partitionState: mutable.Map[TopicAndPartition,
```

```
    PartitionState] = mutable.Map.empty
  ......
}
```

其中分区状态由 PartitionState 表示，内部通过一个字节表示不同的状态，如下所示：

```
sealed trait PartitionState { def state: Byte }
case object NewPartition extends PartitionState { val state: Byte = 0 }
case object OnlinePartition extends PartitionState { val state: Byte = 1 }
case object OfflinePartition extends PartitionState { val state: Byte = 2 }
case object NonExistentPartition extends PartitionState { val state: Byte = 3 }
```

可见分区状态主要有 4 种，分别为 NonExistentPartition、NewPartition、OnlinePartition、OfflinePartition，其各自的生命周期如图 5-4 所示。

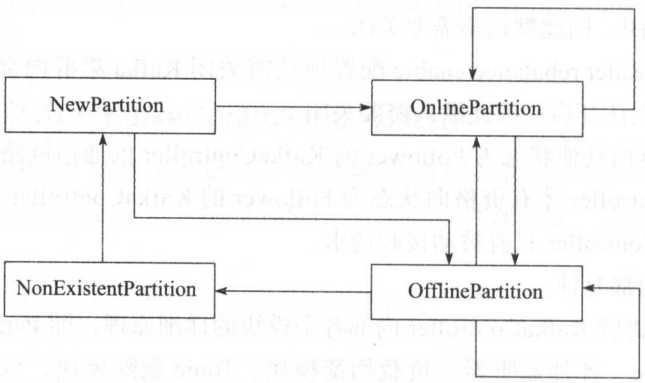


图 5-4 分区状态的生命周期

其中 NonExistentPartition 代表分区从来没有被创建或者被创建之后又被删除的状态；NewPartition 代表分区刚创建，并且包含了 AR，但是此时 Leader 或者 ISR 还没有创建，也就是此时分区还是非活动状态，无法接收数据；OnlinePartition 代表分区的 Leader 已经被选举出来，并且此时已经产生了对应的 ISR，也就是此时分区是活动状态，可以接收数据；OfflinePartition 代表了分区的 Leader 由于某种原因下线时导致分区暂时不可用的状态。

5.3.2 分区状态的转换

分区状态之间的转换是有一定规则的，不是任意状态都可以随便转换，每个目标状态都是由一个合理的前置状态转换而来，状态的转换规则如表 5-1 所示。

表 5-1 分区状态转换规则

目标状态	前置状态	场　景
NewPartition	NonExistentPartition	用户通过 Kafka 客户端将 Topic 的基本信息写入 Zookeeper，KafkaController 监听到 /brokers/topics 目录上数据发生变化，加载新创建的 Topic 的基本信息，包括分区个数、AR 列表

（续）

目标状态	前置状态	场　景
OnlinePartition	NewPartition OnlinePartition OfflinePartition	1）针对新创建的分区选举出 Leader 和生成 ISR 列表，默认为 AR 列表 2）针对分区重新进行 Leader 的选举和生成新的 ISR 列表
OfflinePartition	NewPartition OnlinePartition OfflinePartition	AR 列表中没有任何在线的 Broker Server
NonExistentPartition	OfflinePartition	分区被删除

PartitionStateMachine 内部的 handleStateChange 负责分区状态的具体转换逻辑，handleStateChange 函数的具体参数如下所示：

```
private def handleStateChange(topic: String,
                              partition: Int,
                              targetState: PartitionState,
                              leaderSelector: PartitionLeaderSelector,
                              callbacks: Callbacks) {
  ......
}
```

其中 topic 表明具体的某个 Topic，partition 代表分区索引，targetState 代表目标状态，leaderSelector 代表该分区上 Leader Replica 选举器，针对不同场景有不同的选举策略，目前一共有 5 种选举器，分别为 OfflinePartitionLeaderSelector、ReassignedPartitionLeaderSelector、PreferredReplicaPartitionLeaderSelector、ControlledShutdownLeaderSelector、NoOpLeaderSelector，callbacks 为回调函数，暂时没用到。

接下来将先详细描述状态之间转换的逻辑，由于 handleStateChange 函数本身比较长，因此针对不同的状态转换流程只截取其相关的处理逻辑。

5.3.2.1　NonExistentPartition -> NewPartition

转换代码如下：

```
case NewPartition =>
  // 校验前置状态是否为 NonExistentPartition
  assertValidPreviousStates(topicAndPartition,
                            List(NonExistentPartition),
                            NewPartition)
  /* 从 Zookeeper 目录上 / /brokers/topics/ 具体的 Topic 读取该 Topic 的 AR 列表，并且保存
     至 KafkaController 内存 */
  assignReplicasToPartitions(topic, partition)
  // 将其状态切换成 NewPartition
  partitionState.put(topicAndPartition, NewPartition)
  val assignedReplicas = controllerContext.partitionReplicaAssignment
    (topicAndPartition).mkString(",")
  stateChangeLogger.trace("Controller %d epoch %d changed partition %s state
```

```
        from %s to %s with assigned replicas %s".format(controllerId, controller.
        epoch, topicAndPartition, currState, targetState, assignedReplicas)
```

可见 NonExistentPartition 切换为 NewPartition 的逻辑比较简单，仅仅是持久化该 Topic 在 Zookeeper 目录上的各个分区 AR 列表至 KafkaController 内存，然后置分区状态为 NewPartition。

5.3.2.2 NewPartition、OnlinePartition、OfflinePartition-> OnlinePartition

转换代码如下：

```
case OnlinePartition =>
  //校验前置状态是否为 NewPartition、OnlinePartition、OfflinePartition
  assertValidPreviousStates(topicAndPartition,
                            List(NewPartition, OnlinePartition,
                            OfflinePartition),OnlinePartition)
  partitionState(topicAndPartition) match {
    case NewPartition =>
      //利用分区的 AR 列表初始化 Leader 和 ISR
      initializeLeaderAndIsrForPartition(topicAndPartition)
    case OfflinePartition =>
      //利用 Leader Replica 选举器来初始化 Leader 和 ISR
      electLeaderForPartition(topic, partition, leaderSelector)
    case OnlinePartition => //invoked when the leader needs to be re-elected
      //利用 Leader Replica 选举器来初始化 Leader 和 ISR
      electLeaderForPartition(topic, partition, leaderSelector)
    case _ => //should never come here since illegal previous states are checked above
  }
  //只要确定了分区的 Leader 和 ISR，则置分区状态为 OnlinePartition，此时可以接收数据了
  partitionState.put(topicAndPartition, OnlinePartition)
  val leader = controllerContext.partitionLeadershipInfo(topicAndPartition).
    leaderAndIsr.leader
  stateChangeLogger.trace("Controller %d epoch %d changed partition %s from
    %s to %s with leader %d".format(controllerId, controller.epoch,
    topicAndPartition, currState, targetState, leader))
```

可见在 NewPartition 转换为 OnlinePartition 的过程中并没有需要利用 Leader Replica 选举器来进行 Leader 和 ISR 的选择，那是因为 NewPartition 转换为 OnlinePartition 属于分区刚被创建且准备上线的时候，此时分区没有包含任何数据且 AR 列表在集群中是均衡分布的，因此默认 AR 列表中第一个 Live Broker 为其 Leader，并且 Live Broker 为其 ISR，其 initializeLeaderAndIsrForPartition 的具体流程如下所示：

```
private def initializeLeaderAndIsrForPartition(topicAndPartition:
  TopicAndPartition) {
  //获取分区的 AR 列表
  val replicaAssignment = controllerContext.partitionReplicaAssignment(topicAn
    dPartition)
  //剔除 AR 列表中非在线的 Replica
  val liveAssignedReplicas = replicaAssignment.filter(
```

```
  r => controllerContext.liveBrokerIds.contains(r))
liveAssignedReplicas.size match {
  case 0 =>
    //AR列表中的所有Broker Server都下线，此时无法转换状态
    val failMsg = ("encountered error during state change of partition %s
                   from New to Online, assigned replicas are [%s], " +
                   "live brokers are [%s]. No assigned replica is alive.")
                   .format(topicAndPartition, replicaAssignment.
                   mkString(","),controllerContext.liveBrokerIds)
    throw new StateChangeFailedException(failMsg)
  case _ =>
    //选取第一个作为Leader，所有live Replica为ISR
    val leader = liveAssignedReplicas.head
    //形成分区的LeaderIsrAndControllerEpoch信息
    val leaderIsrAndControllerEpoch = new LeaderIsrAndControllerEpoch(
     new LeaderAndIsr(leader, liveAssignedReplicas.toList),
     controller.epoch)
    try {
      //持久化至Zookeeper
      ZkUtils.createPersistentPath(
        controllerContext.zkClient,
        ZkUtils.getTopicPartitionLeaderAndIsrPath(
          topicAndPartition.topic,
          topicAndPartition.partition),
        ZkUtils.leaderAndIsrZkData(
          leaderIsrAndControllerEpoch.leaderAndIsr,
          controller.epoch))
      //更新至KafkaController的内存
      controllerContext.partitionLeadershipInfo.put(
        topicAndPartition,
        leaderIsrAndControllerEpoch)
      //组装元数据请求同步本节点的信息给集群剩余的Broker Server
      brokerRequestBatch.addLeaderAndIsrRequestForBrokers(
        liveAssignedReplicas,
        topicAndPartition.topic,
        topicAndPartition.partition,
        leaderIsrAndControllerEpoch,
        replicaAssignment)
    } catch {
      case e: ZkNodeExistsException =>
        //Zookeeper节点已存在，转换失败
        val leaderIsrAndEpoch = ReplicationUtils.getLeaderIsrAndEpochForPartition(
          zkClient,
          topicAndPartition.topic,
          topicAndPartition.partition).get
        val failMsg = ("encountered error while changing partition %s's state
                       from New to Online since LeaderAndIsr path already " +
                       "exists with value %s and controller epoch %d")
                       .format(topicAndPartition,
                               leaderIsrAndEpoch.leaderAndIsr.toString(),
                               leaderIsrAndEpoch.controllerEpoch)
```

```
            throw new StateChangeFailedException(failMsg)
        }
    }
}
```

但是在 OfflinePartition 和 OnlinePartition 转换为 OnlinePartition 的过程中需要利用 Leader Replica 选举器来进行 Leader 和 ISR 的选择，那是因为 OfflinePartition 转换为 OnlinePartition 主要发生在分区的所有副本下线又部分上线的时候，此时 Leader 和 ISR 的选择需要考虑初始的 ISR、AR 和当前 Live Broker 列表之间的关系，优先选择 ISR 列表中第一个 Live Broker，否则选择 AR 列表中第一个 Live Broker；OnlinePartition 转换为 OnlinePartition 主要发生在当前集群 Topic 分区的 Leader 分布不均衡，导致节点和节点之间负载波动比较大时，这会触发分区的 Leader Replica 重新选举，需要优先考虑 AR 列表中的第一个 Live Replica 作为其 Leader Replica，或者发生在当前 Leader Replica 准备下线，此时会触发分区的 Leader Replica 重新选举使分区重新上线，需要优先考虑 ISR 列表中排在原始 Leader 之后的 Replica 作为新的 Leader。由于不同的场景 Leader 选举的策略是不一样的，所以需要提供针对不同场景的 PartitionLeaderSelector，但是其 electLeaderForPartition 过程都是一样的，如下所示：

```
def electLeaderForPartition(topic: String,
                            partition: Int,
                            leaderSelector: PartitionLeaderSelector) {
  //组装 TopicAndPartition
  val topicAndPartition = TopicAndPartition (topic, partition)
  try {
    var zookeeperPathUpdateSucceeded: Boolean = false
    var newLeaderAndIsr: LeaderAndIsr = null
    var replicasForThisPartition: Seq[Int] = Seq.empty[Int]
    while(!zookeeperPathUpdateSucceeded) {
      //从 Zookeeper 读取 LeaderIsrAndControllerEpoch
      val currentLeaderIsrAndEpoch = getLeaderIsrAndEpochOrThrowException(
        topic,
        partition)
      val currentLeaderAndIsr = currentLeaderIsrAndEpoch.leaderAndIsr
      val controllerEpoch = currentLeaderIsrAndEpoch.controllerEpoch
      if (controllerEpoch > controller.epoch) {
        /*
         * KafkaController 只有状态为 Leader 才能触发 Partition Leader 选举
         * 如果 Zookeeper 上记录的 controllerEpoch 大于当前的 epoch，则表明当前的
         * KafkaController 已经过时了
         */
        throw new StateChangeFailedException(failMsg)
      }
      //根据 TopicAndPartition 和当前的 LeaderAndIsr 选举出新的 LeaderAndIsr
      val (leaderAndIsr, replicas) = leaderSelector.selectLeader(
        topicAndPartition,
        currentLeaderAndIsr)
```

```
        //持久化至 Zookeeper
        val (updateSucceeded, newVersion) = ReplicationUtils.updateLeaderAndIsr(
             zkClient,
             topic,
             partition,
             leaderAndIsr,
             controller.epoch,
             currentLeaderAndIsr.zkVersion)
        newLeaderAndIsr = leaderAndIsr
        newLeaderAndIsr.zkVersion = newVersion
        zookeeperPathUpdateSucceeded = updateSucceeded
        replicasForThisPartition = replicas
      }
      val newLeaderIsrAndControllerEpoch = new LeaderIsrAndControllerEpoch
        (newLeaderAndIsr,
        controller.epoch)
      //更新至 KafkaController 的内存
      controllerContext.partitionLeadershipInfo.put(
        TopicAndPartition(topic, partition),
        newLeaderIsrAndControllerEpoch)
      val replicas = controllerContext.partitionReplicaAssignment(
        TopicAndPartition(topic, partition))
      //组装元数据请求同步本节点的信息给集群剩余的 Broker Server
      brokerRequestBatch.addLeaderAndIsrRequestForBrokers(
        replicasForThisPartition,
        topic,
        partition,
        newLeaderIsrAndControllerEpoch, replicas)
    } catch {
      case lenne: LeaderElectionNotNeededException =>
      case nroe: NoReplicaOnlineException => throw nroe
      case sce: Throwable =>
        throw new StateChangeFailedException(failMsg, sce)
    }
  }
```

5.3.2.3　NewPartition、OnlinePartition、OfflinePartition-> OfflinePartition

转换代码如下：

```
case OfflinePartition =>
  //校验前置状态是否为 NewPartition、OnlinePartition、OfflinePartition
  assertValidPreviousStates(
    topicAndPartition,
    List(NewPartition, OnlinePartition, OfflinePartition),
    OfflinePartition)
  //将其状态转换成 OfflinePartition
  partitionState.put(topicAndPartition, OfflinePartition)
```

可见 NewPartition、OnlinePartition 和 OfflinePartition 转换为 NewPartition 的逻辑比较简单，仅仅是在 KafkaController 内存中置分区状态为 OfflinePartition。

5.3.2.4 OfflinePartition-> NonExistentPartition

转换代码如下：

```
case NonExistentPartition =>
  //校验前置状态是否为 OfflinePartition
  assertValidPreviousStates(
    topicAndPartition,
    List(OfflinePartition),
    NonExistentPartition)
  //将其状态转换成 NonExistentPartition
  partitionState.put(topicAndPartition, NonExistentPartition)
```

可见 OfflinePartition 转换为 NonExistentPartition 的逻辑比较简单，仅仅是在 KafkaController 内存中将分区状态置为 NonExistentPartition。

5.3.3 PartitionStateMachine 模块的启动

在 onControllerFailover 中会触发 PartitionStateMachine 的启动，PartitionStateMachine 在启动过程中会初始化各个 Partition 的状态，首先会根据 Leader Replica 是否在线初始化为 OnlinePartition 或者 OfflinePartition，其次由于没有被分配 Leader Replica，因此被初始化为 NewPartition，接着尝试将状态 OfflinePartition 或者 NewPartition 的 Partition 转换为 OnlinePartition，最后将 Partition 的状态通过 ControllerChannelManager 同步给其他剩余的 Broker Server。

PartitionStateMachine 详细的启动流程如下：

```
def startup() {
  //初始化分区状态
  initializePartitionState()
  hasStarted.set(true)
  //触发 OnlinePartition 状态的转换
  triggerOnlinePartitionStateChange()
  info("Started partition state machine with initial state -> " + partitionState.
    toString())
}
```

其中 initializePartitionState 会将分区初始化为三种状态：NewPartition、OnlinePartition、OfflinePartition，其具体实现过程如下：

```
private def initializePartitionState() {
  for((topicPartition,replicaAssignment) <- controllerContext.partitionReplicaAssignment) {
    controllerContext.partitionLeadershipInfo.get(topicPartition) match {
      case Some(currentLeaderIsrAndEpoch) =>
        //Partition 已经被分配了 Leader 和 ISR
        controllerContext.liveBrokerIds.contains(
          currentLeaderIsrAndEpoch.leaderAndIsr.leader) match {
          case true =>
            //Leader Replica 所在的 Broker Server 当前在线
```

```
            partitionState.put(topicPartition, OnlinePartition)
          case false =>
            // Leader Replica 所在的 Broker Server 当前离线
            partitionState.put(topicPartition, OfflinePartition)
        }
      case None =>
        // Partition 没有被分配 Leader 和 ISR
        partitionState.put(topicPartition, NewPartition)
    }
  }
}
```

当区分出 NewPartition、OnlinePartition、OfflinePartition 三种状态之后，会尝试将 NewPartition 和 OfflinePartition 转换为 OnlinePartition，其具体实现过程如下：

```
def triggerOnlinePartitionStateChange() {
  try {
    brokerRequestBatch.newBatch()
    // 剔除处于删除状态的 Topic
    for((topicAndPartition, partitionState) <- partitionState
      if(!controller.deleteTopicManager.isTopicQueuedUpForDeletion(
        topicAndPartition.topic))) {
        /* 筛选出 OfflinePartition 和 NewPartition 状态的分区，然后努力将其转换为
          OnlinePartition*/
       if(partitionState.equals(OfflinePartition) || partitionState.equals
        (NewPartition))
        handleStateChange(topicAndPartition.topic,
                          topicAndPartition.partition,
                          OnlinePartition,
                          controller.offlinePartitionSelector,
                          (new CallbackBuilder).build)
    }
    // 同步分区信息给其他的 Broker Server
    brokerRequestBatch.sendRequestsToBrokers(
      controller.epoch,
      controllerContext.correlationId.getAndIncrement)
  } catch {
    case e: Throwable => error("Error while moving some partitions to the
      online state", e)
  }
}
```

5.4　Topic 分区的领导者副本选举策略

Topic 分区的 Leader Replica 在不同场景下的选举策略是不一样的，不同的选举策略都继承某个特征类，即 PartitionLeaderSelector，代码如下：

```
trait PartitionLeaderSelector {
  def selectLeader(topicAndPartition: TopicAndPartition,
```

```
                currentLeaderAndIsr: LeaderAndIsr): (LeaderAndIsr, Seq[Int])
}
```

PartitionLeaderSelector 根据 Topic、Partition、当前的 Leader、当前的 ISR 选举出新的 Leader、新的 ISR 和新的 AR（在线状态）。

目前一共有 5 种不同的策略，其对应的使用场景如表 5-2 所示。

表 5-2 Leader Replica 选举策略

选举策略名称	使用场景
NoOpLeaderSelector	默认的选举策略
ReassignedPartitionLeaderSelector	当分区的 AR 重新分配时使用的策略
PreferredReplicaPartitionLeaderSelector	集群内部自动平衡负载或者用户触发手动平衡负载时使用的策略
OfflinePartitionLeaderSelector	分区状态从 OfflinePartition 或者 NewPartition 切换为 OnlinePartition 时使用的策略
ControlledShutdownLeaderSelector	Leader 状态的 KafkaController 处理其他 Broker Server 下线导致分区的 Leader Replica 发生切换时使用的策略

接下来将对这 5 种不同的策略详细阐述内部的实现原理。

5.4.1 NoOpLeaderSelector

该策略为 KafkaController 内部默认的分区副本选举策略，它其实什么都没做，仅仅是返回当前的 Leader、ISR 和 AR，其实现如下所示：

```
class NoOpLeaderSelector(controllerContext: ControllerContext)
  extends PartitionLeaderSelector with Logging {
  def selectLeader(topicAndPartition: TopicAndPartition,
                   currentLeaderAndIsr: LeaderAndIsr): (LeaderAndIsr, Seq[Int]) = {
    warn("I should never have been asked to perform leader election, returning
      the current LeaderAndIsr and replica assignment.")
    // 仅仅是将输入参数 currentLeaderAndIsr 返回，以及对应 topicAndPartition 的 AR 列表
    (currentLeaderAndIsr, controllerContext.partitionReplicaAssignment
      (topicAndPartition))
  }
}
```

5.4.2 OfflinePartitionLeaderSelector

当 KafkaController 尝试将分区状态从 OfflinePartition 或者 NewPartition 切换为 OnlinePartition 的时候会使用这种策略。OfflinePartitionLeaderSelector 会按照以下步骤来进行 Leader Replica 的选举：

1）筛选出在线的 ISR 和在线的 AR。

2）优先在在线的 ISR 中选择，在线的 ISR 列表不为空，则选择在线 ISR 列表中的第一个，结束选举。

3）在线的 ISR 为空，则根据 unclean.leader.election.enable 的配置是否在在线的 AR 列表中选择，unclean.leader.election.enable 代表了是否允许不在 ISR 列表中选举 Leader，默认为 true，如果设置为 true，则选择在线的 AR 列表中的第一个，结束选举；如果 AR 列表为空，则选举失败。

OfflinePartitionLeaderSelector 的大致实现过程如下：

```
class OfflinePartitionLeaderSelector(controllerContext: ControllerContext,
                                     config: KafkaConfig)
  extends PartitionLeaderSelector with Logging {
  def selectLeader(topicAndPartition: TopicAndPartition,
                   currentLeaderAndIsr: LeaderAndIsr): (LeaderAndIsr, Seq[Int]) = {
    controllerContext.partitionReplicaAssignment.get(topicAndPartition) match {
      // AR 列表存在
      case Some(assignedReplicas) =>
        // 筛选出在线的 AR 列表
        val liveAssignedReplicas = assignedReplicas.filter(
          r => controllerContext.liveBrokerIds.contains(r))
        // 筛选出在线的 ISR 列表
        val liveBrokersInIsr = currentLeaderAndIsr.isr.filter(
          r => controllerContext.liveBrokerIds.contains(r))
        // 获取当前的选举时钟，在每次成功选举之后自增
        val currentLeaderEpoch = currentLeaderAndIsr.leaderEpoch
        val currentLeaderIsrZkPathVersion = currentLeaderAndIsr.zkVersion
        val newLeaderAndIsr = liveBrokersInIsr.isEmpty match {
          // 在线的 ISR 列表为空
          case true =>
            /* 判断是否允许在非 ISR 列表中选择 Leader，因为在非 ISR 列表中选择 Leader 会存在
               丢失数据的风险 */
            if (!LogConfig.fromProps(config.props.props,AdminUtils.fetchTopic
                                     Config(controllerContext.zkClient,
                                            topicAndPartition.topic)).
                                            uncleanLeader-ElectionEnable) {
              // 不允许，则选举失败
              throw new NoReplicaOnline-Exception(("No broker in ISR for partition " +
                "%s is alive. Live brokers are: [%s],".format(topicAndPartition,
                controllerContext.liveBrokerIds)) +
                "ISR brokers are: [%s]".format(currentLeaderAndIsr.isr.mk-String(",")))
            }
            // 允许，则在在线的 AR 列表中选举
            liveAssignedReplicas.isEmpty match {
              case true =>
                // 在线的 AR 列表为空，则选举失败
                throw new NoReplicaOnlineException(("No replica for partition " +
                  "%s is alive. Live brokers are: [%s],".format(topicAndPartition,
                  controllerContext.liveBrokerIds)) +
                  " Assigned replicas are: [%s]".format(assignedReplicas))
              case false =>
                // 在线的 AR 列表不为空，则选择第一个 Replica 作为 Leader
                ControllerStats.uncleanLeaderElectionRate.mark()
```

```
                val newLeader = liveAssignedReplicas.head
                new LeaderAndIsr(newLeader,
                                 currentLeaderEpoch + 1,
                                 List(newLeader),
                                 currentLeaderIsrZkPathVersion + 1)
              }
            case false =>
              // 在线的 ISR 列表不为空，则选择第一个 Replica 作为 Leader
              val liveReplicasInIsr = liveAssignedReplicas.filter(r =>
                liveBrokersInIsr.contains(r))
              val newLeader = liveReplicasInIsr.head
              new LeaderAndIsr(newLeader,
                               currentLeaderEpoch + 1,
                               liveBrokersInIsr.toList,
                               currentLeaderIsrZkPathVersion + 1)
          }
          (newLeaderAndIsr, liveAssignedReplicas)
        case None =>
          // 分区暂时还没有 AR，即没有给分区分配 Replica，则选举异常
          throw new NoReplicaOnlineException("Partition %s doesn't have replicas
                                             assigned to it".format
                                             (topicAndPartition))
      }
    }
  }
```

5.4.3 ReassignedPartitionLeaderSelector

随着 Topic 的新建和删除以及 Broker Server 的上下线，原本 Topic 分区的 AR 列表在集群中的分布变得越来越不均匀了，此时如果管理员下发分区重分配的指令，就会在 Zookeeper 的 /admin/reassign_partitions 目录下指定 Topic 分区的 AR 列表，此时 Leader 状态的 KafkaController 监测到这个路径的数据发生变化，就会触发相应的回调函数，促使对应的 Topic 分区发生 Leader Replica 的选举。

ReassignedPartitionLeaderSelector 会按照以下步骤来进行 Leader Replica 的选举：

1）获取指定的 AR 列表。

2）针对指定的 AR 列表，在线的 Broker Server 和当前的 ISR 列表求交集。

3）如果交集不为空，则选举成功，其第一个 Replica 即为新的 Leader；否则选举失败。

ReassignedPartitionLeaderSelector 的大致实现过程如下：

```
class ReassignedPartitionLeaderSelector(controllerContext: ControllerContext)
  extends PartitionLeaderSelector with Logging {
  def selectLeader(topicAndPartition: TopicAndPartition,
                   currentLeaderAndIsr: LeaderAndIsr): (LeaderAndIsr, Seq[Int]) = {
    // 获取指定的 AR 列表
    val reassignedInSyncReplicas =
      controllerContext.partitionsBeingReassigned(topicAndPartition).newReplicas
```

```
        //获取当前的选举时钟，在每次成功选举之后自增
        val currentLeaderEpoch = currentLeaderAndIsr.leaderEpoch
        val currentLeaderIsrZkPathVersion = currentLeaderAndIsr.zkVersion
        //在指定的 AR 列表中过滤出同时位于在线的 Broker 列表和原始 ISR 列表中的 Replica
        val aliveReassignedInSyncReplicas = reassignedInSyncReplicas.filter(
          r => controllerContext.liveBrokerIds.contains(r) &&
          currentLeaderAndIsr.isr.contains(r))
        //获取筛选结果的第一个值
        val newLeaderOpt = aliveReassignedInSyncReplicas.headOption
        newLeaderOpt match {
          //如果第一个值存在，则选举为 Leader，选举成功
          case Some(newLeader) => (new LeaderAndIsr(newLeader,
                                                    currentLeaderEpoch + 1,
                                                    currentLeaderAndIsr.isr,
                                                    currentLeaderIsrZkPathVersion + 1),
                                   reassignedInSyncReplicas)
          //第一个值不存在，即无法筛选出满足条件的 Replica，选举失败
          case None =>
            reassignedInSyncReplicas.size match {
              case 0 =>
                throw new NoReplicaOnlineException("List of reassigned replicas for
                  partition " +
                  " %s is empty. Current leader and ISR: [%s]".format(topicAndPartition,
                    currentLeaderAndIsr))
              case _ =>
                throw new NoReplicaOnlineException("None of the reassigned replicas
                  for partition " +
                  "%s are in-sync with the leader. Current leader and ISR: [%s]".
                    format(topicAndPartition, currentLeaderAndIsr))
            }
        }
      }
    }
```

5.4.4　PreferredReplicaPartitionLeaderSelector

随着 Topic 的新建和删除以及 Broker Server 的上下线，原本 Topic 分区的 Leader Replica 在集群中的分布变得越来越不均匀了；如果配置 auto.leader.rebalance.enable 为 true，则会自动触发分区的 Leader Replica 选举，或者管理员下发分区 Leader Replica 的选举指令。这会在 Zookeeper 的 /admin/preferred_replica_election 指定具体的 Topic 和分区，此时 Leader 状态的 KafkaController 监测到这个路径的数据发生变化，就会触发相应的回调函数，促使对应的 Topic 分区发生 Leader Replica 的选举。

PreferredReplicaPartitionLeaderSelector 会按照以下步骤来进行 Leader Replica 的选举：

1）获取分区原始的 AR 列表。

2）选择第一个 Replica 作为待定的 Leader Replica。

3）判断待定的 Leader Replica 是否在在线的 Broker Server 中和当前的 ISR 列表中，如

果是，则选举成功，其第一个 Replica 即为新的 Leader；否则选举失败。

PreferredReplicaPartitionLeaderSelector 的大致实现过程如下：

```
class PreferredReplicaPartitionLeaderSelector(controllerContext:
  ControllerContext)
  extends PartitionLeaderSelector with Logging {
  def selectLeader(topicAndPartition: TopicAndPartition,
                   currentLeaderAndIsr: LeaderAndIsr): (LeaderAndIsr, Seq[Int]) = {
    // 获取当前的 AR 列表
    val assignedReplicas = controllerContext.partitionReplicaAssignment
      (topicAndPartition)
    // 选择第一个 Replica 作为待定的 Leader Replica
    val preferredReplica = assignedReplicas.head
    val currentLeader = controllerContext.partitionLeadershipInfo(topicAndPartition)
      .leaderAndIsr.leader
    if (currentLeader == preferredReplica) {
      // 如果当前的 Leader 就是 AR 列表中的第一个 Replica，则不需要均衡
      throw new LeaderElectionNotNeededException(
        "Preferred replica %d is already the current leader for partition %s"
        .format(preferredReplica, topicAndPartition))
    } else {
      if (controllerContext.liveBrokerIds.contains(preferredReplica)
        && currentLeaderAndIsr.isr.contains(preferredReplica)) {
        // 待定的 Leader Replica 位于在线的 Broker Server 中和当前的 ISR 列表中，选举成功
        (new LeaderAndIsr(preferredReplica,
                          currentLeaderAndIsr.leaderEpoch + 1,
                          currentLeaderAndIsr.isr,
                          currentLeaderAndIsr.zkVersion + 1),
                          assignedReplicas)
      } else {
        /* 待定的 Leader Replica 没有位于在线的 Broker Server 中和当前的 ISR 列表中，选举
          失败 */
        throw new StateChangeFailedException("Preferred replica %d for partition"
          .format(preferredReplica) +
          "%s is either not alive or not in the isr. Current leader and ISR:
          [%s]".format(topicAndPartition, currentLeaderAndIsr))
      }
    }
  }
}
```

5.4.5 ControlledShutdownLeaderSelector

当 Broker Server 下线的时候会向 Leader 状态的 KafkaController 下发 ControlledShutdown-Request 的指令，KafkaController 接收到该指令之后会针对位于该 Broker Server 上的 Leader Replica 的分区重新进行 Leader Replica 选举。

ControlledShutdownLeaderSelector 会按照以下步骤来进行 Leader Replica 的选举：

1）获取分区的 ISR 列表。

2）在 ISR 列表中剔除离线的 Replica 作为新的 ISR 列表。

3）如果新的 ISR 列表不为空，则选举成功，其第一个 Replica 即为新的 Leader；否则选举失败。

ControlledShutdownLeaderSelector 的大致实现过程如下：

```
class ControlledShutdownLeaderSelector(controllerContext: ControllerContext)
  extends PartitionLeaderSelector with Logging {
  def selectLeader(topicAndPartition: TopicAndPartition,
                   currentLeaderAndIsr: LeaderAndIsr): (LeaderAndIsr, Seq[Int]) = {
    //获取当前的选举时钟，在每次成功选举之后自增
    val currentLeaderEpoch = currentLeaderAndIsr.leaderEpoch
    val currentLeaderIsrZkPathVersion = currentLeaderAndIsr.zkVersion
    //获取当前的 Leader
    val currentLeader = currentLeaderAndIsr.leader
    //获取当前的 AR
    val assignedReplicas = controllerContext.partitionReplicaAssignment
      (topicAndPartition)
    //获取在线的 Broker Server 列表
    val liveOrShuttingDownBrokerIds = controllerContext.
      liveOrShuttingDownBrokerIds
    //筛选出在线的 AR 列表
    val liveAssignedReplicas = assignedReplicas.filter(
      r => liveOrShuttingDownBrokerIds.contains(r))
    //在当前的 ISR 列表中剔除准备下线的 Broker Server
    val newIsr = currentLeaderAndIsr.isr.filter(
      brokerId => !controllerContext.shuttingDownBrokerIds.contains(brokerId))
    //取结果的第一个 Replica
    val newLeaderOpt = newIsr.headOption
      newLeaderOpt match {
      //第一个 Replica 存在，则选举成功
        case Some(newLeader) =>
          (LeaderAndIsr(newLeader,
                        currentLeaderEpoch + 1,
                        newIsr,
                        currentLeaderIsrZkPathVersion + 1),
           liveAssignedReplicas)
      //第一个 Replica 不存在，即当前的 ISR 已经全部下线，则选举失败
      case None =>
        throw new StateChangeFailedException(("No other replicas in ISR %s for
          %s besides" +
          " shutting down brokers %s").format(currentLeaderAndIsr.isr.mkString(","),
          topicAndPartition, controllerContext.shuttingDownBrokerIds.mkString(",")))
    }
  }
}
```

5.5　Topic 分区的副本状态转换机制

Topic 分区的副本状态维护是由 ReplicaStateMachine 模块负责的，Topic 分区的副本状态伴随着 Topic 分区状态的变化而变化。

5.5.1 副本状态的分类

ReplicaStateMachine 内部的 replicaState 变量保存了每个具体的 Topic 分区副本的状态，即

```
class ReplicaStateMachine (controller: KafkaController) extends Logging {
  ......
  private val replicaState: mutable.Map[PartitionAndReplica, ReplicaState] =
    mutable.Map.empty
  ......
}
```

其中分区副本状态由 ReplicaState 表示，内部通过一个字节表示不同的状态，如下所示：

```
sealed trait ReplicaState { def state: Byte }
case object NewReplica extends ReplicaState { val state: Byte = 1 }
case object OnlineReplica extends ReplicaState { val state: Byte = 2 }
case object OfflineReplica extends ReplicaState { val state: Byte = 3 }
case object ReplicaDeletionStarted extends ReplicaState { val state: Byte = 4}
case object ReplicaDeletionSuccessful extends ReplicaState { val state: Byte = 5}
case object ReplicaDeletionIneligible extends ReplicaState { val state: Byte = 6}
case object NonExistentReplica extends ReplicaState { val state: Byte = 7 }
```

可见分区副本状态主要有 7 种，分别为 NewReplica、OnlineReplica、OfflineReplica、ReplicaDeletionStarted、ReplicaDeletionSuccessful、ReplicaDeletionIneligible 和 NonExistentReplica，其各自的生命周期如图 5-5 所示。

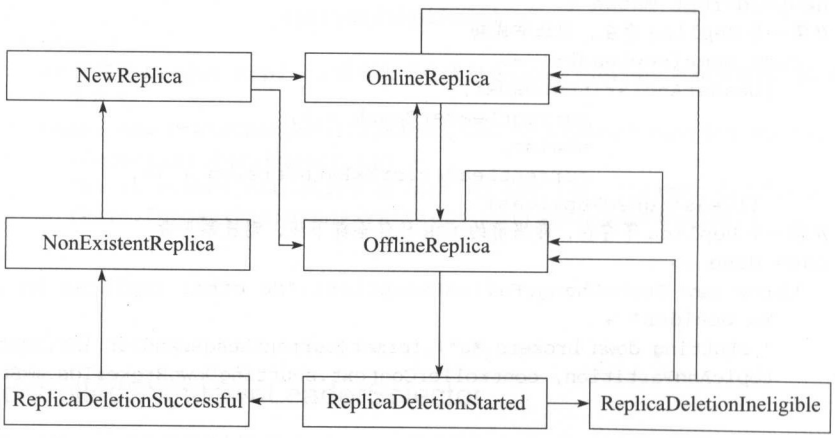


图 5-5 分区副本状态的生命周期

其中 NonExistentReplica 代表分区副本被彻底删除之后的状态；NewReplica 代表分区副本刚被分配，但是还没有开始工作时候的状态；OnlineReplica 代表分区副本开始工作时的状态，此时该副本是该分区的 Leader 或者 Follower；OfflineReplica 代表分区副本所在的 Broker Server 宕机时所导致的副本状态；ReplicaDeletionStarted 代表分区副本下线之后准备

开始删除的状态；ReplicaDeletionSuccessful 代表了相关的 Broker Server 正确响应分区副本被删除请求之后的状态；ReplicaDeletionIneligible 代表了相关的 Broker Server 错误响应分区副本被删除请求之后的状态。

5.5.2　副本状态的转换

分区副本状态之间的转换是有一定的规则的，不是可以随便转换任意状态，每个目标状态都是由一个合理的前置状态转换而来的，状态的转换规则如表 5-3 所示。

表 5-3　分区副本状态转换规则

目标状态	前置状态	场　景
NewReplica	NonExistentReplica	1）用户通过 Kafka 客户端将 Topic 的基本信息写入 Zookeeper，KafkaController 监听到 /brokers/topics 目录上数据发生变化，加载新创建的 Topic 的基本信息，包括分区个数、AR 列表，并且将分区副本状态转换为 NewReplica，但是此时分区上的 Leader Replica 还没有选举出来 2）用户通过 Kafka 客户端将 Topic 重分区的相关信息写入 Zookeeper，KafkaController 监听到 /admin/reassign_partitions 目录上数据发生变化，加载新的 Topic 分区信息，并且将分区副本状态转换为 NewReplica，但是此时该分区上的 Leader Replica 还没有选举出来
OnlineReplica	NewReplica OnlineReplica OfflineReplica ReplicaDeletionIneligible	1）KafkaController 在成为 Leader 的时候会尝试将位于在线 Broker Server 上的 Replica 状态转换为 OnlineReplica 2）KafkaController 针对新创建的 Topic 分区进行 Leader Replica 选举，选举成功之后将对应的分区副本状态转换为 OnlineReplica 3）KafkaController 针对重分配的 Topic 分区进行 Leader Replic 选举，选举成功之后将对应的分区副本状态转换为 OnlineReplica 4）KafkaController 监听到 /brokers/ids 目录上数据发生变化，将原本位于新上线的 Broker Server 上的 Replica 状态转换为 OnlineReplica
OfflineReplica	NewReplica OnlineReplica OfflineReplicaReplicaDeletionIneligible	1）KafkaController 在成为 Leader 的时候会尝试将位于离线 Broker Server 上的 Replica 状态转换为 OfflineReplica 2）KafkaController 接收到 Broker Server 的 ControlledShutdownRequest 时会尝试将对应 Broker Server 上的 Replica 状态转换为 OfflineReplica 3）KafkaController 针对重分配的 Topic 分区进行 Leader Replic 选举，选举成功之后立刻将之前旧的 Replica 状态转换为 OfflineReplica 4）用户通过 Kafka 客户端将准备删除的 Topic 写入 Zookeeper，KafkaController 监听到 /admin/ delete_topics 目录上数据发生变化，读取 Topic 信息，将 Topic 的所有 Replica 状态转换为 OfflineReplica

（续）

目标状态	前置状态	场　景
ReplicaDeletionStarted	OfflineReplica	1）KafkaController 针对重分配的 Topic 分区进行 Leader Replic 选举，选举成功之后准备删除旧的 Replica 的时候会将 Replica 状态转换为 ReplicaDeletionStarted 2）用户通过 Kafka 客户端将准备删除的 Topic 写入 Zookeeper，KafkaController 监听到 /admin/ delete_topics 目录上数据发生变化，读取 Topic 信息，当准备开始删除 Replica 的时候会将 Replica 状态转换为 ReplicaDeletionStarted
ReplicaDeletionSuccessful	ReplicaDeletionStarted	KafkaController 针对 Topic 的 Replica 被删除成功的情况会将 Replica 的状态转换为 ReplicaDeletionSuccessful
ReplicaDeletionIneligible	ReplicaDeletionStarted	KafkaController 针对 Topic 的 Replica 被删除失败的情况会将 Replica 的状态转换为 ReplicaDeletionIneligible
NonExistentReplica	ReplicaDeletionSuccessful	KafkaController 针对 Topic 的 Replica 被删除成功的情况最后将 Replica 的状态从 ReplicaDeletionSuccessful 转换为 NonExistentReplica

ReplicaStateMachine 内部的 handleStateChange 负责了分区副本状态的具体转换逻辑，其 handleStateChange 函数的具体参数如下所示：

```
def handleStateChange(partitionAndReplica: PartitionAndReplica,
                      targetState: ReplicaState,
                      callbacks: Callbacks) {
  ......
}
```

其中 partitionAndReplica 标识了具体的 Topic、Partition 和 Replica，targetState 代表目标状态，callbacks 为回调函数。

接下来将先详细描述状态之间转换的逻辑，由于 handleStateChange 函数本身比较长，因此针对不同的状态转换流程只截取部分处理逻辑。

5.5.2.1 NonExistentReplica-> NewReplica

转换代码如下：

```
case NewReplica =>
  //校验前置状态是否为 NonExistentReplica
  assertValidPreviousStates(partitionAndReplica,List(NonExistentReplica), targetState)
  //获取 Replica 所在分区的 Leader, ISR, LeaderEpoch, ControllerEpoch
  val leaderIsrAndControllerEpochOpt = ReplicationUtils.getLeaderIsrAndEpochFo
    rPartition(zkClient, topic, partition)
  leaderIsrAndControllerEpochOpt match {
    //上述信息存在，说明该分区已经选举出 Leader
    case Some(leaderIsrAndControllerEpoch) =>
      if(leaderIsrAndControllerEpoch.leaderAndIsr.leader == replicaId)
        //Leader 不可能是准备状态转换的 Replica
        throw new StateChangeFailedException("Replica %d for partition %s cannot
```

```
            be moved to NewReplica"
            .formatreplicaId, topicAndPartition) + "state as it is being requested
              to become leader")
          /* 否则向 ReplicaId 所在的 Broker Server 发送 LeaderAndIsrRequest 请求，同步 Topic
            元数据 */
          brokerRequestBatch.addLeaderAndIsrRequestForBrokers(
            List(replicaId),
            topic,
            partition,
            leaderIsrAndControllerEpoch,
            replicaAssignment)
        case None =>
      }
      // 将其状态转换成 NewReplica
      replicaState.put(partitionAndReplica, NewReplica)
```

NonExistentReplica 转换为 NewReplica 的逻辑比较简单，首先确保对应分区的 Leader 不是该 Replica，然后发送 LeaderAndIsrRequest 更新该 Replica 所在 Broker Server 上的有关 Topic 的元数据，最后置分区状态为 NewReplica。

5.5.2.2 NewReplica、OnlineReplica、OfflineReplica、ReplicaDeletionIneligible -> OnlineReplica

转换代码如下：

```
case OnlineReplica =>
  /* 校验前置状态是否为 NewReplica、OnlineReplica、OfflineReplica
  ReplicaDeletionIneligible*/
  assertValidPreviousStates(partitionAndReplica,
                            List(NewReplica,
                                 OnlineReplica,
                                 OfflineReplica,
                                 ReplicaDeletionIneligible),
                            targetState)
  replicaState(partitionAndReplica) match {
    // 前置状态为 NewReplica，则说明是刚创建的
    case NewReplica =>
      val currentAssignedReplicas = controllerContext.partitionReplicaAssignment
        (topicAndPartition)
      // 判断当前的 AR 列表是否包含该 ReplicaId，如果不包含，则添加进该分区的 AR 列表
      if(!currentAssignedReplicas.contains(replicaId))
        controllerContext.partitionReplicaAssignment.put(
          topicAndPartition,
          currentAssignedReplicas :+ replicaId)
  // 前置状态为 OnlineReplica、OfflineReplica、ReplicaDeletionIneligible
  case _ =>
    controllerContext.partitionLeadershipInfo.get(topicAndPartition) match {
      case Some(leaderIsrAndControllerEpoch) =>
        /* 向 ReplicaId 所在的 Broker Server 发送 LeaderAndIsrRequest 请求，同步 Topic 元数据 */
        brokerRequestBatch.addLeaderAndIsrRequestForBrokers(
```

```
            List(replicaId),
            topic,
            partition,
            leaderIsrAndControllerEpoch,
            replicaAssignment)
          // 将其状态转换成 OnlineReplica
          replicaState.put(partitionAndReplica, OnlineReplica)
        case None =>
      }
    }
    // 将其状态转换成 OnlineReplica
    replicaState.put(partitionAndReplica, OnlineReplica)
```

NewReplica、OnlineReplica、OfflineReplica 和 ReplicaDeletionIneligible 转换成 OnlineReplica 主要区分两种情况：如果前置状态为 NewReplica，则说明该 Replica 是新建的，因此只要将其添加进该分区的 AR 列表即可；如果前置状态为 OnlineReplica、OfflineReplica 和 ReplicaDeletionIneligible，则需要向该 ReplicaId 所在的 Broker Server 发送 LeaderAndIsrRequest 请求，同步该分区的元数据信息，最后都会将 Replica 状态转换成 OnlineReplica。

5.5.2.3 NewReplica、OnlineReplica、OfflineReplica、ReplicaDeletionIneligible -> OfflineReplica

转换代码如下：

```
case OfflineReplica =>
  /* 校验前置状态是否为 NewReplica, OnlineReplica, OfflineReplica,
    ReplicaDeletionIneligible*/
  assertValidPreviousStates(partitionAndReplica,
                            List(NewReplica,
                                 OnlineReplica,
                                 OfflineReplica,
                                 ReplicaDeletionIneligible),
                            targetState)
  /* 向 ReplicaId 所在的 Broker Server 发送 StopReplicaRequest 请求，其中
    deletePartition 为 false，暂停 Replica 的一切行动 */
  brokerRequestBatch.addStopReplicaRequestForBrokers(
    List(replicaId),
    topic,
    partition,
    updatedLeaderIsrAndControllerEpoch,
    deletePartition = false)
  val leaderAndIsrIsEmpty: Boolean =
    controllerContext.partitionLeadershipInfo.get(topicAndPartition) match {
      // 获取 ReplicaId 所在分区的 Leader, ISR, LeaderEpoch 和 ControllerEpoch
      case Some(currLeaderIsrAndControllerEpoch) =>
        // 将 ReplicaId 从 ISR 中剔除
        controller.removeReplicaFromIsr(topic, partition, replicaId) match {
          // ISR 不为空
          case Some(updatedLeaderIsrAndControllerEpoch) =>
```

```
        // 获取当前的 AR 列表
        val currentAssignedReplicas = controllerContext.partitionReplicaAssi
          gnment(topicAndPartition)
        if (!controller.deleteTopicManager.isPartitionToBeDeleted(topicAndPa
          rtition)) {
        /* 排除正在删除的分区，否则向该分区副本所在的所有 Broker Server 发送
          LeaderAndIsrRequest 请求，同步 Topic 元数据 */
          brokerRequestBatch.addLeaderAndIsrRequestForBrokers(
            currentAssignedReplicas.filterNot(_ == replicaId),
            topic,
            partition,
            updatedLeaderIsrAndControllerEpoch,
            replicaAssignment)
        }
        // 将其状态转换成 OfflineReplica
        replicaState.put(partitionAndReplica, OfflineReplica)
        false
      case None =>
        // 更新之后分区的 Leader、ISR、LeaderEpoch 和 ControllerEpoch 不存在
        true
      }
    case None =>
      // 分区的初始 Leader、ISR、LeaderEpoch 和 ControllerEpoch 不存在
      true
    }
  /* 无论是 Replica 下线前还是下线后，其对应分区的 Leader、ISR、LeaderEpoch 和 ControllerEpoch
    为空，抛异常 */
  if (leaderAndIsrIsEmpty)
    throw new StateChangeFailedException(
    "Failed to change state of replica %d for partition %s since the leader
      and isr path in zookeeper is empty"
    .format(replicaId, topicAndPartition))
```

NewReplica、OnlineReplica、OfflineReplica 和 ReplicaDeletionIneligible 转换成 OfflineReplica 的时候，首先需要向该 ReplicaId 所在的 Broker Server 发送 StopReplicaRequest 请求，其中 deletePartition 为 false，并不删除数据，只是暂停 Replica 的一切行动，其次需要向该分区副本除了指定 ReplicaId 所在的 Broker Server 之外的其他 Broker Server 发送 LeaderAndIsrRequest 请求，同步该分区的元数据。

5.5.2.4　OfflineReplica -> ReplicaDeletionStarted

转换代码如下：

```
case ReplicaDeletionStarted =>
  // 校验前置状态是否为 OfflineReplica
  assertValidPreviousStates(partitionAndReplica, List(OfflineReplica), targetState)
  // 将其状态转换成 OfflineReplica
  replicaState.put(partitionAndReplica, ReplicaDeletionStarted)
  // 向 ReplicaId 所在的 Broker Server 发送 StopReplicaRequest，其中 deletePartition 为 true
```

```
brokerRequestBatch.addStopReplicaRequestForBrokers(
  List(replicaId),
  topic,
  partition,
  deletePartition = true,
  callbacks.stopReplicaResponseCallback)
```

OfflineReplica 转换为 ReplicaDeletionStarted 的逻辑比较简单，首先置分区状态为 ReplicaDeletionStarted，然后向 ReplicaId 所在的 Broker Server 发送 StopReplicaRequest，其中 deletePartition 为 true，真正删除数据。

5.5.2.5 ReplicaDeletionStarted -> ReplicaDeletionSuccessful

转换代码如下：

```
case ReplicaDeletionSuccessful =>
  // 校验前置状态是否为 ReplicaDeletionStarted
  assertValidPreviousStates(partitionAndReplica,
                            List(ReplicaDeletionStarted),
                            targetState)
  // 置分区状态为 ReplicaDeletionSuccessful
  replicaState.put(partitionAndReplica, ReplicaDeletionSuccessful)
```

ReplicaDeletionStarted 转换为 ReplicaDeletionSuccessful 的逻辑比较简单，就是将分区状态置为 ReplicaDeletionSuccessful。

5.5.2.6 ReplicaDeletionStarted -> ReplicaDeletionIneligible

转换代码如下：

```
case ReplicaDeletionIneligible =>
  // 校验前置状态是否为 ReplicaDeletionStarted
  assertValidPreviousStates(partitionAndReplica,
                            List(ReplicaDeletionStarted),
                            targetState)
  // 置分区状态为 ReplicaDeletionIneligible
  replicaState.put(partitionAndReplica, ReplicaDeletionIneligible)
```

ReplicaDeletionStarted 转换为 ReplicaDeletionIneligible 的逻辑比较简单，就是将分区状态置为 ReplicaDeletionIneligible。

5.5.2.7 ReplicaDeletionSuccessful -> NonExistentReplica

转换代码如下：

```
case NonExistentReplica =>
  // 校验前置状态是否为 ReplicaDeletionSuccessful
  assertValidPreviousStates(partitionAndReplica,
                            List(ReplicaDeletionSuccessful),
                            targetState)
  val currentAssignedReplicas = controllerContext.partitionReplicaAssignment
```

```
    (topicAndPartition)
  // 将 ReplicaId 从 AR 列表中剔除
  controllerContext.partitionReplicaAssignment.put(
    topicAndPartition,
    currentAssignedReplicas.filterNot(_ == replicaId))
  // 彻底删除 Replica 状态
  replicaState.remove(partitionAndReplica)
```

ReplicaDeletionSuccessful 转换为 NonExistentReplica 的时候，首先将该 ReplicaId 从分区的 AR 列表中剔除，然后将该 ReplicaId 从 KafkaController 的内存中直接删除。

5.5.3　ReplicaStateMachine 模块的启动

在 onControllerFailover 中会触发 ReplicaStateMachine 模块的启动，ReplicaStateMachine 在启动过程中会初始化各个 Replica 的状态，首先会根据 Replica 是否在线初始化为 OnlineReplica 或者 ReplicaDeletionIneligible，然后将 OnlineReplica 状态的 Replica 转换为 OnlineReplica 状态，最后将 Replica 的状态通过 ControllerChannelManager 同步给其他剩余的 Broker Server。

ReplicaStateMachine 详细的启动流程如下：

```
def startup() {
  // 初始化 Replica 状态
  initializeReplicaState()
  hasStarted.set(true)
  // 触发 OnlineReplica 状态的转换
  handleStateChanges(controllerContext.allLiveReplicas(), OnlineReplica)
  info("Started replica state machine with initial state -> " + replicaState.
    toString())
}
```

其中 initializeReplicaState 会将 Replica 初始化为两种状态：OnlineReplica 和 ReplicaDeletionIneligible，其具体实现过程如下：

```
private def initializeReplicaState() {
  // 遍历 Topic 的 Partition 列表
  for((topicPartition, assignedReplicas) <- controllerContext.
    partitionReplicaAssignment) {
    val topic = topicPartition.topic
    val partition = topicPartition.partition
    // 遍历 Partition 的 AR 列表
    assignedReplicas.foreach { replicaId =>
      val partitionAndReplica = PartitionAndReplica(topic, partition, replicaId)
      controllerContext.liveBrokerIds.contains(replicaId) match {
        /* 该 replicaId 所在的 Broker Server 位于在线状态，将 replica 状态初始化为
          OnlineReplica*/
        case true => replicaState.put(partitionAndReplica, OnlineReplica)
        /* 该 replicaId 所在的 Broker Server 位于离线状态，将 replica 状态初始化为
          ReplicaDeletionIneligible*/
```

```
            case false =>
              replicaState.put(partitionAndReplica, ReplicaDeletionIneligible)
          }
        }
      }
    }
```

当区分出 OnlineReplica 和 ReplicaDeletionIneligible 两种状态之后，会尝试将 OnlineReplica 转换为 OnlineReplica 状态，并且同步 Replica 信息给相关的 Broker Server，其具体实现过程如下：

```
def handleStateChanges(replicas: Set[PartitionAndReplica],
                       targetState: ReplicaState,
                       callbacks: Callbacks = (new CallbackBuilder).build) {
  if(replicas.size > 0) {
    info("Invoking state change to %s for replicas %s".format(targetState, replicas.
      mkString(",")))
    try {
      brokerRequestBatch.newBatch()
      /* 通过 controllerContext.allLiveReplicas() 筛选出在线状态的 Replica，然后将其转
        换为 OnlineReplica*/
      replicas.foreach(r => handleStateChange(r, targetState, callbacks))
      // 同步 Replica 信息给其他的 Broker Server
      brokerRequestBatch.sendRequestsToBrokers(
        controller.epoch,
        controllerContext.correlationId.getAndIncrement)
    }catch {
      case e: Throwable => error("Error while moving some replicas to %s state".
        format(targetState), e)
    }
  }
}
```

5.6 KafkaController 内部的监听器

在 5.2.1 节中的图 5-3 展示了 KafkaController 内监听函数的分布情况，这些监听函数主要是用来维护 Kafka 集群的元数据，例如：在线的 Broker Server 列表、Topic 列表、Partition 对应的 AR 列表，Partition 对应的 ISR 列表，等等。KafkaController 把这些信息都保存在 ControllerContext 里面，ControllerContext 内部保存这些信息的变量如下：

```
class ControllerContext(val zkClient: ZkClient,
                        val zkSessionTimeout: Int) {
  ......
  var shuttingDownBrokerIds: mutable.Set[Int] = mutable.Set.empty
  var allTopics: Set[String] = Set.empty
  var partitionReplicaAssignment: mutable.Map[TopicAndPartition, Seq[Int]] =
    mutable.Map.empty
```

```
  var partitionLeadershipInfo: mutable.Map[TopicAndPartition,
    LeaderIsrAndControllerEpoch] = mutable.Map.empty
  var partitionsBeingReassigned: mutable.Map[TopicAndPartition,
    ReassignedPartitionsContext] = new mutable.HashMap
  var partitionsUndergoingPreferredReplicaElection: mutable.
    Set[TopicAndPartition] = new mutable.HashSet
  private var liveBrokersUnderlying: Set[Broker] = Set.empty
  private var liveBrokerIdsUnderlying: Set[Int] = Set.empty
  ......
}
```

以上各变量保存的内容如表 5-4 所示。

表 5-4　变量保存的内容

变 量 名	类　型	含　义
shuttingDownBrokerIds	Set[Int]	处于关机状态的 Broker 列表，存储的是 Broker Server 的 ID 集合
liveBrokersUnderlying	Set[Broker]	处于在线状态的 Broker 列表，存储的是 Broker Server 的 Broker 对象
liveBrokerIdsUnderlying	Set[Int]	处于在线状态的 Broker 列表，存储的是 Broker Server 的 ID 集合
allTopics	Set[String]	当前 Topic 列表
partitionReplicaAssignment	.Map[TopicAndPartition, Seq[Int]]	当前 TopicAndPartition 的 AR 列表
partitionLeadershipInfo	Map[TopicAndPartition, LeaderIsrAndControllerEpoch]	当前 TopicAndPartition 的 Leader 和 ISR 列表等
partitionsBeingReassigned	Map[TopicAndPartition, ReassignedPartitionsContext]	处于 TopicAndPartition 重分配状态的 AR 列表
partitionsUndergoingPreferredReplicaElection	Set[TopicAndPartition]	为了负载均衡，处于正在经历首选副本选举状态的 TopicAndPartition 列表

接下来将详细描述注册在 Zookeeper 上不同的监听器。

5.6.1　TopicChangeListener

PartitionStateMachine 在路径为 /broker/topics 的 Zookeeper 上注册了 TopicChangeListener 监听器，该监听器主要用来监听 Topic 的创建，当监听到有新的 Topic 创建的时候，会触发 TopicChangeListener 内部的 handleChildChange 回调函数，其具体实现如下：

```
class TopicChangeListener extends IZkChildListener with Logging {
  def handleChildChange(parentPath : String, children : java.util.
                        List[String]) {
    inLock(controllerContext.controllerLock) {
      if (hasStarted.get) {
        try {
```

```
            //获取 /broker/topics 目录下所有的 Topic
            val currentChildren = {
              import JavaConversions._
              (children: Buffer[String]).toSet
            }
            /* 根据当前 /broker/topics 目录下的 Topic 列表和 ControllerContext 内部的
               allTopics 列表计算出新增的 Tpoic*/
            val newTopics = currentChildren -- controllerContext.allTopics
            /* 根据当前 /broker/topics 目录下的 Topic 列表和 ControllerContext 内部的
               allTopics 列表计算出删除的 Tpoic*/
            val deletedTopics = controllerContext.allTopics - currentChildren
            //更新 allTopics
            controllerContext.allTopics = currentChildren
            //获取新增 Tpoic 的 AR 列表
            val addedPartitionReplicaAssignment = ZkUtils.getReplicaAssignment
              ForTopics(zkClient, newTopics.toSeq)
            //剔除已经删除的 Topic 的 AR 列表
            controllerContext.partitionReplicaAssignment = controllerContext.
              partitionReplicaAssignment.filter(
              p =>!deletedTopics.contains(p._1.topic))
            //将新增 Topic 的 AR 列表更新至 partitionReplicaAssignment
            controllerContext.partitionReplicaAssignment.++=
          (addedPartitionReplicaAssignment)
            if(newTopics.size > 0)
              //开始创建新增 Topic 的分区
              controller.onNewTopicCreation(
                newTopics,
                addedPartitionReplicaAssignment.keySet.toSet)
          } catch {
            case e: Throwable => error("Error while handling new topic", e )
          }
        }
      }
    }
}
```

如果发现 /broker/topics 目录上有新增的 Topic，则会创建新的 Topic，其流程如下：

```
def onNewTopicCreation(topics: Set[String], newPartitions:
  Set[TopicAndPartition]) {
  //在 /brokers/topics/ 具体的 Topic 目录下注册 AddPartitionsListener 监听器
  topics.foreach(topic => partitionStateMachine.registerPartitionChangeListener
    (topic))
  //创建分区
  onNewPartitionCreation(newPartitions)
}
```

可见创建 Topic 除了在特定的 Topic 目录下注册 AddPartitionsListener 监听器之外，最主要的就是进行分区的创建，只要分区创建成功了那么 Topic 也就创建成功了，其分区创建

的流程如下：

```
def onNewPartitionCreation(newPartitions: Set[TopicAndPartition]) {
  // 置分区状态为 NewPartition，主要就是初始化分区的状态
  partitionStateMachine.handleStateChanges(newPartitions, NewPartition)
  // 置副本状态为 NewReplica，主要就是初始化副本的状态
  replicaStateMachine.handleStateChanges(
    controllerContext.replicasForPartition(newPartitions),
    NewReplica)
  /* 置分区状态为 OnlinePartition，其实就是通过 offlinePartitionSelector 选举出
     LeaderReplica*/
  partitionStateMachine.handleStateChanges(
    newPartitions,
    OnlinePartition,
    offlinePartitionSelector)
  // 置副本状态为 OnlineReplica，其实就是添加副本至分区的 AR 列表
  replicaStateMachine.handleStateChanges(
    controllerContext.replicasForPartition(newPartitions),
    OnlineReplica)
}
```

在这里，读者可以回想下之前小节所描述的分区状态和副本状态的转换，Topic 的创建本质上就是分区的创建，分区的创建过程中会伴随着分区状态的转换和分区副本状态的转换。

5.6.2　AddPartitionsListener

在 Topic 的创建过程中会在 /brokers/topics/[topic] 目录下注册 AddPartitionsListener 监听器，该监听器主要用来监听 Topic 分区的变化。当监听到有新的分区创建的时候，会触发 AddPartitionsListener 内部的 handleDataChange 回调函数，其具体实现如下：

```
class AddPartitionsListener(topic: String) extends IZkDataListener with
  Logging {
  def handleDataChange(dataPath : String, data: Object) {
    inLock(controllerContext.controllerLock) {
      try {
        // 获取该 topic 的分区列表
        val partitionReplicaAssignment =
          ZkUtils.getReplicaAssignmentForTopics(zkClient, List(topic))
        /* 通过 ControllerContext 内部的 partitionReplicaAssignment 筛选处于新增的分区
           列表 */
        val partitionsToBeAdded = partitionReplicaAssignment.filter(
          p =>!controllerContext.partitionReplicaAssignment.contains(p._1))
        if(controller.deleteTopicManager.isTopicQueuedUpForDeletion(topic))
          // 忽略正在删除状态中的 topic
          error("Skipping adding partitions %s for topic %s since it is
            currently being deleted"
```

```
                .format(partitionsToBeAdded.map(_._1.partition).mkString(","), topic))
            else {
              if (partitionsToBeAdded.size > 0) {
                //如果新增的分区列表大于 0，则进行分区的创建
                controller.onNewPartitionCreation(partitionsToBeAdded.keySet.toSet)
              }
            }
          } catch {
            case e: Throwable => error("Error while handling add partitions for
                                       data path " + dataPath, e )
          }
        }
      }
    }
```

如果发现有新增的分区时就会进行分区的创建，其创建流程和创建 Topic 的时候一样，读者可以参考 5.6.1 节。

5.6.3 PartitionsReassignedListener

KafkaControl 转换为 Leader 的过程中在路径为 /admin/reassign_partitions 的 Zookeeper 上注册了 PartitionsReassignedListener 监听器，该监听器主要用来监听 Topic 分区的重分配。当监听到有新的 Topic 分区需要重分配的时候。会触发 PartitionsReassignedListener 内部的 handleDataChange 回调函数，其具体实现如下：

```
class PartitionsReassignedListener(controller: KafkaController)
  extends IZkDataListener with Logging {
  def handleDataChange(dataPath: String, data: Object) {
    //获取 /admin/reassign_partitions 目录下所有待重分配的分区
    val partitionsReassignmentData = ZkUtils.parsePartitionReassignmentData
      (data.toString)
    /* 通过 ControllerContext 内部的 partitionsBeingReassigned 剔除已经处于重分配
      状态的分区 */
    val partitionsToBeReassigned = inLock(controllerContext.controllerLock) {
      partitionsReassignmentData.filterNot(
        p => controllerContext.partitionsBeingReassigned.contains(p._1))
    }
    partitionsToBeReassigned.foreach {
      partitionToBeReassigned => inLock(controllerContext.controllerLock) {
        if(controller.deleteTopicManager.isTopicQueuedUpForDeletion(
          partitionToBeReassigned._1.topic)) {
          /*topic 分区处于删除状态，因此删除在 /admin/reassign_partitions 和
              ControllerContext 内的 partitionsBeingReassigned 的记录 */
          controller.removePartitionFromReassignedPartitions(partitionToBeReassigned._1)
        } else {
          /* 由于分区的重分配涉及副本数据的同步，不是立刻可以完成，因此需要初始化相关信息 */
          val context = new ReassignedPartitionsContext(partitionToBeReassign
```

```
            ed._2)
          controller.initiateReassignReplicasForTopicPartition(
            partitionToBeReassigned._1,
            context)
          }
        }
      }
    }
  }
```

由于分区的重分配涉及副本数据的同步，不是立刻可以转换完成，需要一定的时间，因此需要一些初始化步骤来启动分区的重分配过程，其初始化步骤如下：

```
def initiateReassignReplicasForTopicPartition(
  topicAndPartition: TopicAndPartition,
  reassignedPartitionContext: ReassignedPartitionsContext) {
  //重分配分区的 AR 列表
  val newReplicas = reassignedPartitionContext.newReplicas
  val topic = topicAndPartition.topic
  val partition = topicAndPartition.partition
  //过滤出在线的重分配分区的 AR 列表
  val aliveNewReplicas = newReplicas.filter(
    r => controllerContext.liveBrokerIds.contains(r))
  try {
    //获取重分配分区当前的 AR 列表
    val assignedReplicasOpt = controllerContext.partitionReplicaAssignment.
      get(topicAndPartition)
    assignedReplicasOpt match {
      case Some(assignedReplicas) =>
        if(assignedReplicas == newReplicas) {
          //当前 AR 列表和待重分配的 AR 列表相同，则说明不需要进行重分配
          throw new KafkaException("Partition %s to be reassigned is already
            assigned to replicas".format(topicAndPartition) +
            " %s. Ignoring request for partition reassignment".
              format(newReplicas.mkString(",")))
        } else {
          if(aliveNewReplicas == newReplicas) {
          //重分配分区的 AR 列表中其 Replica 所在的 Broker Server 都在线
          /* 在 /brokers/topics/[topic]/partitions/partitionId/state/ 目录上注册
            ReassignedPartitionsIsrChangeListener */
          watchIsrChangesForReassignedPartition(
            topic,
            partition,
            reassignedPartitionContext)
          //将该分区转换为正在进行重分配状态
          controllerContext.partitionsBeingReassigned.put(
            topicAndPartition,
            reassignedPartitionContext)
```

```
                //标记该 Topic 暂时无法删除
                deleteTopicManager.markTopicIneligibleForDeletion(Set(topic))
                //正式启动分区重分配
                onPartitionReassignment(topicAndPartition,
                  reassignedPartitionContext)
              } else {
                //重分配分区的 AR 列表中其 Replica 所在的 Broker Server 并没有都在线
                throw new KafkaException("Only %s replicas out of the new set of
                  replicas".format(aliveNewReplicas.mkString(",")) +
                  " %s for partition %s to be reassigned are alive.
                  ".format(newReplicas.mkString(","), topicAndPartition) +
                  "Failing partition reassignment")
              }
            }
          case None => throw new KafkaException("Attempt to reassign partition %s
            that doesn't exist".format(topicAndPartition))
        }
      } catch {
        case e: Throwable => removePartitionFromReassignedPartitions(topicAndPartition)
      }
    }
```

在正式启动分区的重分配之前会判断是否需要进行重分配，重分配之后的 AR 列表和当前的 AR 列表不相同并且重分配之后的 AR 列表所在的 Broker Server 都在线，满足以上两个条件才会触发分区的重分配。

为了更好的说明分区重分配的流程，先介绍一些缩写词：

- RAR：Reassigned replicas，即分区重分配之后的 AR 列表。
- OAR：Original list of replicas for partition，即分区原始的 AR 列表。

分区的重分配过程主要分为两个阶段：

阶段一： 新副本数据的同步，为了防止数据丢失，必须使新的副本数据和旧的副本数据保持一致。

阶段二： 删除旧的副本数据，更新元数据。一旦新的副本数据和旧的副本数据保持一致之后就可以删除旧的副本数据了。

如果 RAR 列表位于当前的 ISR 列表之中，即此时分区重分配之后的副本数据已经和旧的副本数据保持一致，则直接进入阶段二；如果 RAR 列表没有全部位于当前的 ISR 列表之中，即此时分区重分配之后的副本数据没有和旧的数据保持一致，则进入阶段一。因此着重描述后者情况，即 RAR 列表没有全部位于当前的 ISR 列表之中，前者情况已经包含在后者的第二个阶段中。

假设 Kafka 集群由 3 个 Broker Server 组成，其 Broker Id 分别为 1、2 和 3，并且 Kafka-Controller 的 Leader 为 1。具体的 Topic 为 Topic-0，待重分配的分区索引为 0，当前的 AR 列表为 [1,2]，其中 Leader Replica 为 2，需要重分配到 [1,3]。

阶段一的步骤如图 5-6 所示。

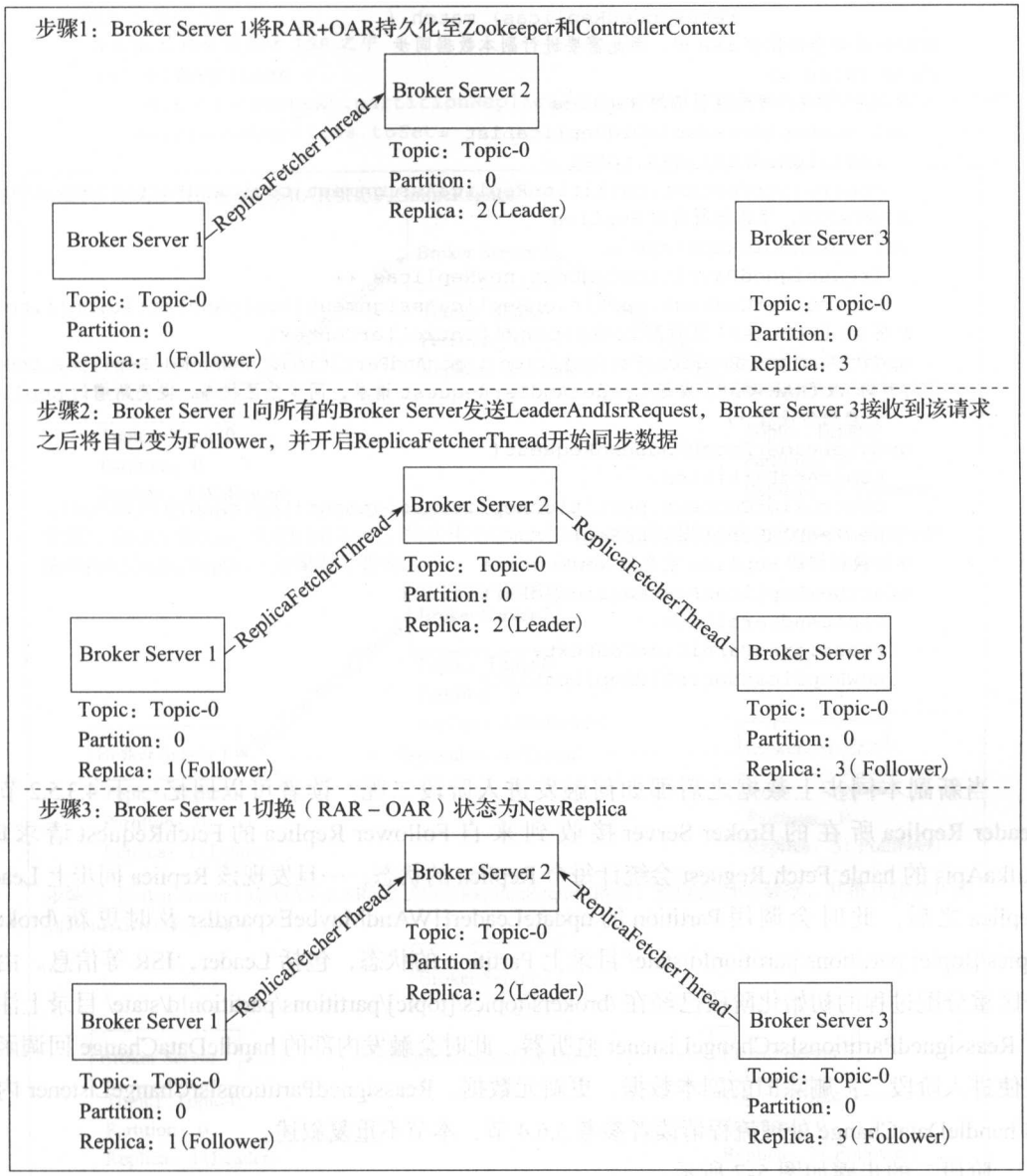


图 5-6　分区重分配阶段一

其对应的代码如下：

```
def onPartitionReassignment(topicAndPartition: TopicAndPartition,
                            reassignedPartitionContext:
                            ReassignedPartitionsContext) {
  val reassignedReplicas = reassignedPartitionContext.newReplicas
  areReplicasInIsr(topicAndPartition.topic,
                   topicAndPartition.partition,
```

```
                    reassignedReplicas) match {
      // RAR 没有全部位于 ISR 中，因此需要进行副本数据同步
      case false =>
        // RAR-OAR，筛选出新增的 Replica
        val newReplicasNotInOldReplicaList =
          reassignedReplicas.toSet -
          controllerContext.partitionReplicaAssignment(topicAndPartition).toSet
        // RAR+OAR，筛选出所有的 Replica
        val newAndOldReplicas =
          (reassignedPartitionContext.newReplicas ++
          controllerContext.partitionReplicaAssignment(topicAndPartition)).toSet
        // 将 AR（RAR+OAR）更新至 Zookeeper 和 ControllerContext
        updateAssignedReplicasForPartition(topicAndPartition, newAndOldReplicas.toSeq)
        /* 向 AR（RAR+OAR）发送 LeaderAndIsrRequest 请求，同步分区状态，促使新增的 Replica
          开启同步线程 */
        updateLeaderEpochAndSendRequest(
          topicAndPartition,
          controllerContext.partitionReplicaAssignment(topicAndPartition),
          newAndOldReplicas.toSeq)
        // 切换新增的 Replica 状态为 NewReplica
        startNewReplicasForReassignedPartition(
          topicAndPartition,
          reassignedPartitionContext,
          newReplicasNotInOldReplicaList)
    ......
  }
```

当新副本同步上数据之后那如何触发进入阶段二呢？读者可以回忆一下 4.3.5.2 节当 Leader Replica 所在的 Broker Server 接收到来自 Follower Replica 的 FetchRequest 请求时，KafkaApis 的 hanle Fetch Reguest 会统计每个 Replica 的状态，一旦发现该 Replica 同步上 Leader Replica 之后，此时会调用 Partition 的 updateLeaderHWAndMaybeExpandIsr 及时更新 /brokers/topics/[topic]/partitions/partitionId/state/ 目录上 Partition 的状态，包括 Leader，ISR 等信息。由于分区重分配过程的初始化阶段已经在 /brokers/topics/[topic]/partitions/partitionId/state/ 目录上注册了 ReassignedPartitionsIsrChangeListener 监听器，此时会触发内部的 handleDataChange 回调函数促使进入阶段二：删除旧的副本数据，更新元数据。ReassignedPartitionsIsrChangeListener 内部的 handleDataChange 处理流程请读者参考 5.6.4 节，本节不重复叙述。

阶段二的步骤如图 5-7 所示。

其对应的代码如下：

```
def onPartitionReassignment(topicAndPartition: TopicAndPartition,
                            reassignedPartitionContext:
                            ReassignedPartitionsContext) {
  val reassignedReplicas = reassignedPartitionContext.newReplicas
  areReplicasInIsr(topicAndPartition.topic,
                   topicAndPartition.partition,
                   reassignedReplicas) match {
    ......
```

```
case true =>
  // RAR 已经全部位于 ISR 之中
  val oldReplicas =
    controllerContext.partitionReplicaAssignment(topicAndPartition).toSet -
    reassignedReplicas.toSet
```

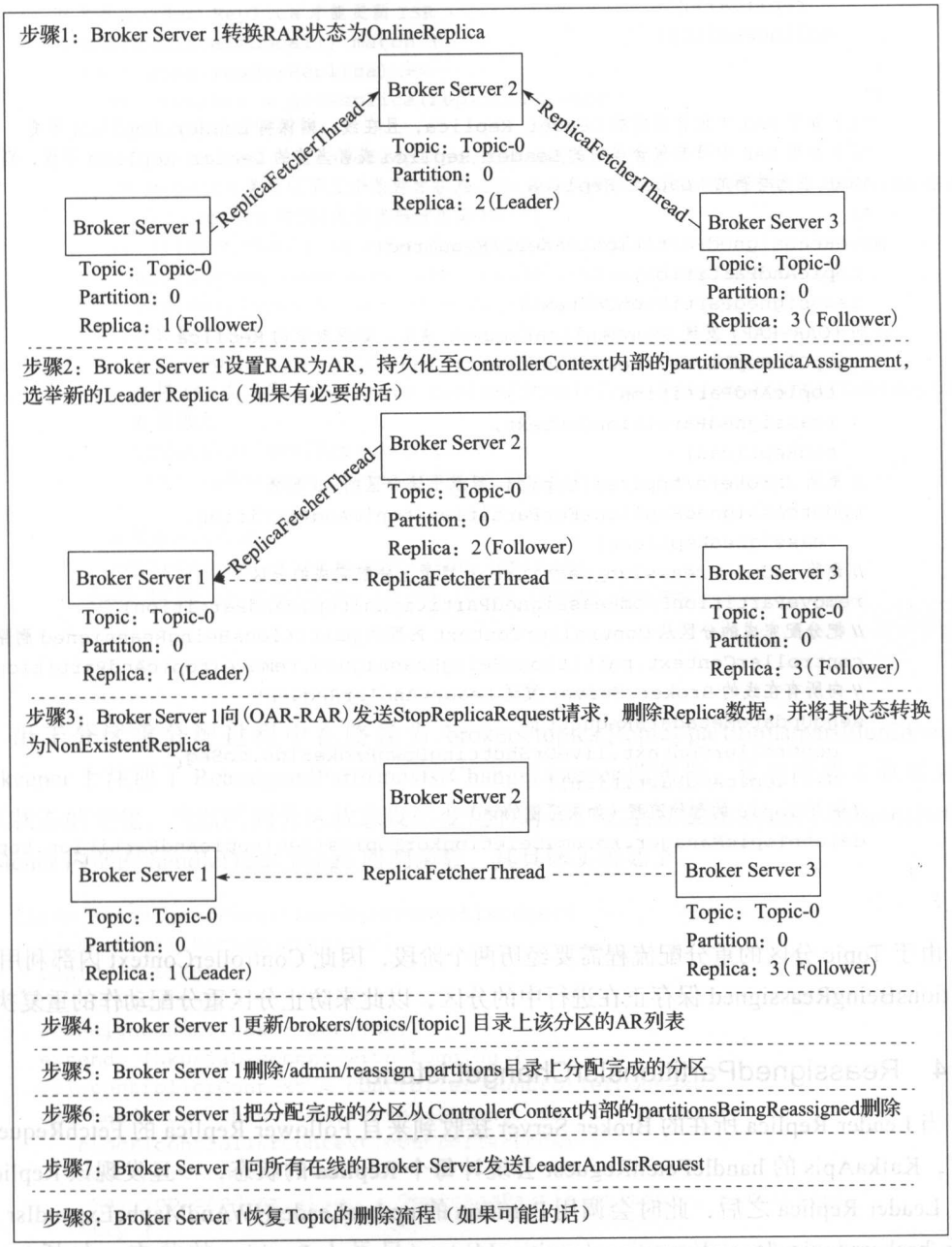


图 5-7　分区重分配阶段二

```
        // 转换 RAR 的状态为 OnlineReplica
        reassignedReplicas.foreach {
          replica => replicaStateMachine.handleStateChanges(
            Set(new PartitionAndReplica(topicAndPartition.topic,
              topicAndPartition.partition,
              replica)),
            OnlineReplica)
        }
        /*
         *1）如果 RAR 中包含当前的 Leader Replica，且在线，则保持 Leader Replica 不变
         *2）如果 RAR 中没有包含当前的 Leader Replica 或者当前的 Leader Replica 下线，则在
         *RAR 中选举新的 Leader Replica
         */
        moveReassignedPartitionLeaderIfRequired(
          topicAndPartition,
          reassignedPartitionContext)
        // 向（OAR-RAR）发送 StopReplicaRequest 请求，删除对应的 Replica
        stopOldReplicasOfReassignedPartition(
          topicAndPartition,
          reassignedPartitionContext,
          oldReplicas)
        // 更新 /brokers/topics/[topic] 目录上该分区的 AR 列表
        updateAssignedReplicasForPartition(topicAndPartition,
          reassignedReplicas)
        // 删除 /admin/reassign_partitions 目录上分配完成的分区
        removePartitionFromReassignedPartitions(topicAndPartition)
        // 把分配完成的分区从 ControllerContext 内部的 partitionsBeingReassigned 删除
        controllerContext.partitionsBeingReassigned.remove(topicAndPartition)
        // 向所有在线的 Broker Server 发送 LeaderAndIsrRequest
        sendUpdateMetadataRequest(
          controllerContext.liveOrShuttingDownBrokerIds.toSeq,
          Set(topicAndPartition))
        // 恢复 Topic 的删除流程（如果可能的话）
        deleteTopicManager.resumeDeletionForTopics(Set(topicAndPartition.topic))
      }
    }
```

由于 Topic 分区的重分配流程需要经历两个阶段，因此 ControllerContext 内部利用变量 partitionsBeingReassigned 保存正在进行中的分区，以此来防止分区重分配动作的重复执行。

5.6.4 ReassignedPartitionsIsrChangeListener

当 Leader Replica 所在的 Broker Server 接收到来自 Follower Replica 的 FetchRequest 请求时，KafkaApis 的 handleFetchReguest 会统计每个 Replica 的状态，一旦发现该 Replica 同步上 Leader Replica 之后，此时会调用 Partition 的 updateLeaderHWAndMaybeExpandIsr 及时更新 /brokers/topics/[topic]/partitions/partitionId/state/ 目录上 Partition 的状态，包括 Leader、

ISR 等信息。

Partition 的 updateLeaderHWAndMaybeExpandIs 具体实现如下：

```
def updateLeaderHWAndMaybeExpandIsr(replicaId: Int) {
  inWriteLock(leaderIsrUpdateLock) {
    // 只有 Leader Replica 才能更新 ISR
    leaderReplicaIfLocal() match {
      case Some(leaderReplica) =>
        val replica = getReplica(replicaId).get
        // 获取该分区当前的 HW
        val leaderHW = leaderReplica.highWatermark
        /* replica 满足以下三个条件才扩展 ISR: 1 )replica 不在 ISR 列表; 2 )replica 在 AR 中;
           3 ) replica 的 HW 大于当前分区的 HW */
        if (!inSyncReplicas.contains(replica) &&
          assignedReplicas.map(_.brokerId).contains(replicaId) &&
          replica.logEndOffset.offsetDiff(leaderHW) >= 0) {
          // 扩展 ISR
          val newInSyncReplicas = inSyncReplicas + replica
          // 将 ISR 信息写入 /brokers/topics/[topic]/partitions/partitionId/state/
          // 目录上
          updateIsr(newInSyncReplicas)
          replicaManager.isrExpandRate.mark()
        }
        // 更新分区的 HW
        maybeIncrementLeaderHW(leaderReplica)
      case None => // nothing to do if no longer leader
    }
  }
}
```

由于分区重分配过程中在路径为 /brokers/topics/[topic]/partitions/partitionId/state/ 的 Zookeeper 上注册了 ReassignedPartitionsIsrChangeListener 监听器，该监听器主要用来监听分区状态的变化，当监听到分区状态发生变化的时候，会触发 ReassignedPartitionsIsrChang eListener 内部的 handleDataChange 回调函数，其具体实现如下：

```
class ReassignedPartitionsIsrChangeListener(
  controller: KafkaController,
  topic: String,
  partition: Int,
  reassignedReplicas : Set[Int])
  extends IZkDataListener with Logging {
  val controllerContext = controller.controllerContext
  def handleDataChange(dataPath: String, data: Object) {
    inLock(controllerContext.controllerLock) {
    // 组装 TopicAndPartition
      val topicAndPartition = TopicAndPartition(topic, partition)
      try {
        // 根据 TopicAndPartition 从 ControllerContext 中获取对应的重分配 AR 列表
```

```
            controllerContext.partitionsBeingReassigned.get(topicAndPartition)
              match {
              case Some(reassignedPartitionContext) =>
                // 获取该分区的 Leader 和 ISR 相关信息
                val newLeaderAndIsrOpt = ZkUtils.getLeaderAndIsrForPartition(
                  zkClient,
                  topic,
                  partition)
                newLeaderAndIsrOpt match {
                  case Some(leaderAndIsr) =>
                    /* 针对 reassignedReplicas 和 ISR 求交集，筛选出已经同步上的重分配 AR 列表 */
                    val caughtUpReplicas = reassignedReplicas & leaderAndIsr.isr.toSet
                    if(caughtUpReplicas == reassignedReplicas) {
                      // 重分配 AR 列表已经全部同步上了，则进入分区重分配的阶段二
                      controller.onPartitionReassignment(
                        topicAndPartition,
                        reassignedPartitionContext)
                    }
                    else {
                      ......
                    }
                  case None => ……
                }
              case None =>
            }
          } catch {
            case e: Throwable => ……
          }
        }
      }
    }
```

因此 ReassignedPartitionsIsrChangeListener 通过观察 /brokers/topics/[topic]/partitions/partitionId/state/ 上分区状态的变化来协助完成分区重分配的整个流程。

5.6.5 PreferredReplicaElectionListener

每一个 Partition 可以有多个 Replica，即 AR 列表。在这个列表中的第一个 Replica 称为“ Preferred Replica”。当创建 Topic 时，Kafka 要确保所有的 Topic 的“ Preferred Replica”均匀地分布在 Kafka 集群中。在理想的场景中，一个 Partition 的初始 Leader Replica 应该是“ Preferred Replica”。这保证了集群所有的 Leader Replica 带来的负载在整个集群是均衡的。然而，如果发生 Broker Shutdown 了 (比如 Crash，机器故障) 的话，那么 Leader Replica 带来的负载就不均衡了。

因此 KafkaController 在切换为 Leader 状态的时候会在 /admin/preferred_replica_election 目录下注册 PreferredReplicaElectionListener 监听器，该监听器主要用来监听哪些 Topic 的

Partition需要重新均衡Leader Replica至Preferred Replica。当监听到此目录下有数据更新时，会触发PreferredReplicaElectionListener内部的handleDataChange回调函数，其具体实现如下：

```
class PreferredReplicaElectionListener(controller: KafkaController)
  extends IZkDataListener with Logging {
  val controllerContext = controller.controllerContext
  def handleDataChange(dataPath: String, data: Object) {
    inLock(controllerContext.controllerLock) {
      //从/admin/preferred_replica_election读取需要重新均衡的Partition
      val partitionsForPreferredReplicaElection = PreferredReplicaLeaderElectionCommand.
        parsePreferredReplicaElectionData(
        data.toString)
      if(controllerContext.partitionsUndergoingPreferredReplicaElection.size >
        0){
        //剔除正在进行均衡的Partition，防止重新均衡
        val partitions = partitionsForPreferredReplicaElection --
          controllerContext.partitionsUndergoingPreferredReplicaElection
        //筛选出正在删除的Partition
        val partitionsForTopicsToBeDeleted = partitions.filter(
          p => controller.deleteTopicManager.isTopicQueuedUpForDeletion(p.topic))
        if(partitionsForTopicsToBeDeleted.size > 0) {
          ......
        }
      }
      /* 针对第一次执行均衡的且其不在删除状态中分区进行Leader Replica的选举，使其切换至
        Preferred Replica*/
      controller.onPreferredReplicaElection(partitions --
        partitionsForTopicsToBeDeleted)
    }
  }
}
```

因为针对分区的重新均衡需要一定的时间，所以需要排除正在进行均衡的分区，防止重复执行。在对合理分区真正执行均衡的时候，本质上就是切换分区的状态，调用不同的分区的领导者副本选举策略对分区副本进行选举，显而易见为了将分区的Leader Replica切换至Preferred Replica，需要调用PreferredReplicaPartitionLeaderSelector选举器。onPreferredReplicaElection的具体流程如下：

```
def onPreferredReplicaElection(
  partitions: Set[TopicAndPartition],
  isTriggeredByAutoRebalance: Boolean = false) {
  try {
    //添加进正在均衡的队伍中
    controllerContext.partitionsUndergoingPreferredReplicaElection ++=
      partitions
    //标记暂时无法被删除
```

```
        deleteTopicManager.markTopicIneligibleForDeletion(partitions.map(_.topic))
        /* 将分区状态切换为 OnlinePartition，利用 preferredReplicaPartitionLeaderSelector
           选举策略进行 Leader Replica 的选举 */
        partitionStateMachine.handleStateChanges(
          partitions,
          OnlinePartition,
          preferredReplicaPartitionLeaderSelector)
      } catch {
        case e: Throwable => error("Error completing preferred replica leader
          election for partitions %s".format(partitions.mkString(",")), e)
      } finally {
        // 清理 /admin/preferred_replica_election 目录和 ControllerContext 的
        // partitionsUndergoingPreferredReplicaElection 中包含的数据
        removePartitionsFromPreferredReplicaElection(partitions,
          isTriggeredByAutoRebalance)
        // 去除标记，表示可以被删除
        deleteTopicManager.resumeDeletionForTopics(partitions.map(_.topic))
      }
    }
```

5.6.6 BrokerChangeListener

Broker Server 启动时会在路径为 /brokers/ids 的 Zookeeper 上创建一个 EphemeralPath（非永久路径），当 Broker Server 由于异常情况导致下线时，此 EphemeralPath 会随着 Broker Server 和 Zookeeper 链接的断开而消失。

由于 Broker Server 的上下线影响着位于该 Broker Server 上所有 Replica 的状态，因此 ReplicaStateMachine 在路径为 /broker/topics 的 Zookeeper 上注册了 BrokerChangeListener 监听器，该监听器主要用来监听 Broker Server 的上下线，当监听到此目录下有数据发生变化的时候，会触发 BrokerChangeListener 内部的 handleChildChange 回调函数，其具体实现如下：

```
  class BrokerChangeListener() extends IZkChildListener with Logging {
    def handleChildChange(parentPath : String,
                          currentBrokerList : java.util.List[String]) {
      inLock(controllerContext.controllerLock) {
        if (hasStarted.get) {
          ControllerStats.leaderElectionTimer.time {
            try {
              // 当前 /brokers/ids 目录下所有的 Broker Server
              val curBrokerIds = currentBrokerList.map(_.toInt).toSet
              // 筛选出新上线的 Broker Server
              val newBrokerIds = curBrokerIds -- controllerContext.
                liveOrShuttingDownBrokerIds
              val newBrokerInfo = newBrokerIds.map(ZkUtils.
                getBrokerInfo(zkClient, _))
              val newBrokers = newBrokerInfo.filter(_.isDefined).map(_.get)
              // 筛选出下线的 Broker Server
```

```
            val deadBrokerIds = controllerContext.liveOrShuttingDownBrokerIds
              - curBrokerIds
            // 更新 ControllerContext 内部在线且不处于正在下线中的 Broker 列表，
            controllerContext.liveBrokers = curBrokerIds.map(
              ZkUtils.getBrokerInfo(zkClient, _)).filter(_.isDefined).map(_.
                get)
            // 建立和新上线的 Broker Server 的通信链路
            newBrokers.foreach(controllerContext.controllerChannelManager.
              addBroker(_))
            // 断开和下线的 Broker Server 的通信链路
          deadBrokerIds.foreach(controllerContext.controllerChannelManager.
            removeBroker(_))
            if(newBrokerIds.size > 0)
              /* 尝试切换位于该 Broker Server 上的 Replica 和 Partition 状态分别至
                OnlineReplica 和 OnlinePartition*/
              controller.onBrokerStartup(newBrokerIds.toSeq)
            if(deadBrokerIds.size > 0)
              /* 尝试切换位于该 Broker Server 上的 Replica 和 Partition 状态分别至
                OfflineReplica 和 OnlinePartition*/
              controller.onBrokerFailure(deadBrokerIds.toSeq)
            } catch {
              case e: Throwable => error("Error while handling broker
                changes", e)
            }
          }
        }
      }
    }
  }
}
```

BrokerChangeListener 主要针对两种情况进行处理：Broker Server 上线的情况，以及下线的情况。

当 Broker Server 上线的时候，主要经历以下几个步骤：

1）Leader 状态的 KafkaController 同步 Kafka 集群所有的 Topic 信息给新上线的 Broker Server。

2）将原本位于该 Broker Server 上的所有 Replica 状态切换至 OnlineReplica。

3）如果 Partition 的 Replica 有且只有一个，并且正好位于该 Broker Server 上，则切换 Partition 状态至 OnlinePartition。

4）如果分区重分配之前由于该 Broker Server 下线导致退出的话，则尝试重新进行分区重分配。

5）如果之前由于 Broker Server 下线导致对应的 Replica 无法删除的话，则恢复删除流程。

其对应的代码如下：

```
def onBrokerStartup(newBrokers: Seq[Int]) {
  val newBrokersSet = newBrokers.toSet
  // 同步集群内的 Topic 信息
  sendUpdateMetadataRequest(newBrokers)
  // 筛选出位于该 Broker Server 上的 Replica
  val allReplicasOnNewBrokers = controllerContext.replicasOnBrokers
    (newBrokerSet)
  // 切换对应 Replica 状态为 OnlineReplica
  replicaStateMachine.handleStateChanges(allReplicasOnNewBrokers,
    OnlineReplica)
  // 尝试将 Partition 状态切换为 OnlinePartition
  partitionStateMachine.triggerOnlinePartitionStateChange()
  // 筛选出需要进行分区重分配的分区
  val partitionsWithReplicasOnNewBrokers = controllerContext.
    partitionsBeingReassigned.filter {
    case (topicAndPartition, reassignmentContext) => reassignmentContext.
      newReplicas.exists(newBrokersSet.contains(_))
  }
  // 继续进行分区重分配
  partitionsWithReplicasOnNewBrokers.foreach(p =>
    onPartitionReassignment(p._1, p._2))
  // 筛选出需要进行删除 Replica 的 Topic
  val replicasForTopicsToBeDeleted = allReplicasOnNewBrokers.filter(
    p => deleteTopicManager.isTopicQueuedUpForDeletion(p.topic))
  // 恢复 Topic 的删除流程
  if(replicasForTopicsToBeDeleted.size > 0) {
    deleteTopicManager.resumeDeletionForTopics(replicasForTopicsToBeDeleted.
      map(_.topic))
  }
}
```

当 Broker Server 下线的时候，主要经历以下几个步骤：

1）更新 ControllerContext 内部正在下线的 Broker Server 列表。

2）将 Leader Replica 位于该 Broker Server 上的分区状态切换为 OfflinePartition，紧接着触发分区状态切换为 OnlinePartition，利用 OfflinePartitionLeaderSelector 副本选举策略进行 Leader Replica 的选举。

3）将位于该 Broker Server 上的 Replica 状态切换为 OfflineReplica。

4）如果对应 Replica 的 Topic 处于删除队列中的话，则标记暂时无法删除。

其对应的代码如下：

```
def onBrokerFailure(deadBrokers: Seq[Int]) {
  // 更新 ControllerContext 内部的 shuttingDownBrokerIds 变量
  val deadBrokersThatWereShuttingDown =
    deadBrokers.filter(id => controllerContext.shuttingDownBrokerIds.
      remove(id))
  val deadBrokersSet = deadBrokers.toSet
```

```
    // 筛选出 Leader Replica 位于该 Broker Server 上的 Partition
    val partitionsWithoutLeader = controllerContext.partitionLeadershipInfo.
      filter(
      partitionAndLeader =>
        deadBrokersSet.contains(partitionAndLeader._2.leaderAndIsr.leader) &&
        !deleteTopicManager.isTopicQueuedUpForDeletion(partitionAndLeader._1.
          topic)
      ).keySet
    // 将 Partition 状态切换为 OfflinePartition
    partitionStateMachine.handleStateChanges(partitionsWithoutLeader,
      OfflinePartition)
    // 尝试将 Partition 再次切换为 OnlinePartition，针对 Leader Replica 重新选举
    partitionStateMachine.triggerOnlinePartitionStateChange()
    // 筛选出位于该 Broker Server 上的 Replica
    var allReplicasOnDeadBrokers = controllerContext.replicasOnBrokers(deadBrokersSet)
    // 剔除准备删除的 Replica
    val activeReplicasOnDeadBrokers = allReplicasOnDeadBrokers.filterNot(
      p => deleteTopicManager.isTopicQueuedUpForDeletion(p.topic))
    // 将剩下的 Replica 状态切换为 OfflineReplica
    replicaStateMachine.handleStateChanges(activeReplicasOnDeadBrokers,
      OfflineReplica)
    // 筛选出准备删除的 Replica
    val replicasForTopicsToBeDeleted = allReplicasOnDeadBrokers.filter(
      p => deleteTopicManager.isTopicQueuedUpForDeletion(p.topic))
    if(replicasForTopicsToBeDeleted.size > 0) {
      /* 标记 Replica 对应的 Topic 暂时无法删除，因为 Broker Server 下线了，无法删除该 Broker
        Server 上的 Replica 数据 */
      deleteTopicManager.failReplicaDeletion(replicasForTopicsToBeDeleted)
    }
  }
```

5.6.7　DeleteTopicsListener

PartitionStateMachine 在路径为 /admin/delete_topics 的 Zookeeper 上注册了 DeleteTopicsListener 监听器，该监听器主要用来监听 Topic 的删除，当监听到有新的 Topic 删除的时候，会触发 DeleteTopicsListener 内部的 handleChildChange 回调函数，其具体实现如下：

```
class DeleteTopicsListener() extends IZkChildListener with Logging {
  def handleChildChange(parentPath : String, children : java.util.
    List[String]) {
    inLock(controllerContext.controllerLock) {
      // 获取 /admin/delete_topics 目录下所有待删除的 Topic
      var topicsToBeDeleted = {
        import JavaConversions._
        (children: Buffer[String]).toSet
      }
      // 通过 ControllerContext 内部的 allTopics 筛选出不存在的 topic
      val nonExistentTopics = topicsToBeDeleted.filter(
```

```
            t => !controllerContext.allTopics.contains(t))
          if(nonExistentTopics.size > 0) {
            //topic 不存在，则删除 zk 上写入的 topic
            nonExistentTopics.foreach(
              topic => ZkUtils.deletePathRecursive(zkClient, ZkUtils.
                getDeleteTopicPath(topic)))
          }
          //剔除不存在的 topic，则剩下的就是需要删除的 topic
          topicsToBeDeleted --= nonExistentTopics
          if(topicsToBeDeleted.size > 0) {
            topicsToBeDeleted.foreach { topic =>
              /* 通过 ControllerContext 内部的 partitionsUndergoingPreferredReplicaEle
                ction 判断该 topic 是否处于首选副本选举过程中 */
              val preferredReplicaElectionInProgress =
                controllerContext.partitionsUndergoingPreferredReplicaElection
                .map(_.topic).contains(topic)
              /* 判断 ControllerContext 内部的 partitionsBeingReassigned 判断该 topic 是
                否处于分区重分配的过程中 */
              val partitionReassignmentInProgress =
                controllerContext.partitionsBeingReassigned.keySet.map(_.topic).
                  contains(topic)
              /* 如果 topic 位于上述其中的某个状态中，则标记为无法删除，需要等待上述状态结束之后
                恢复删除 */
              if(preferredReplicaElectionInProgress || partitionReassignmentInProgress)
                 controller.deleteTopicManager.markTopicIneligibleForDeletion(Set(topic))
            }
            //将待删除的 topic 添加进删除队列
            controller.deleteTopicManager.enqueueTopicsForDeletion(topicsToBeDeleted)
          }
        }
      }
    }
```

DeleteTopicsListener 主要用来负责触发 Topic 的删除，而 TopicDeletionManager 主要用来负责具体 Topic 删除的逻辑，其详细的实现原理可以参考 5.8 小节。

5.7 Kafka 集群的负载均衡流程

在 5.6.5 小节中提到：Partition 的 AR 列表中第一个 Replica 称为“Preferred Replica”，并均匀分布在整个 Kafka 集群中。由于每个 Partition 只有 Leader Replica 对外提供读写服务，并且 Partition 创建的时候默认的 Leader Replica 位于 Preferred Replica 之上，此时 Kafka 集群的负载是均衡的，如果 Kafka 集群长时间运行，Broker Server 中途由于异常而发生重启，此时 Partition 的 Leader Replica 会发生迁移，这样会导致其 Partition 的 Leader Replica 在集群中不在均衡了，因此某些节点的读写压力会明显大于其他节点，如图 5-8 所示。

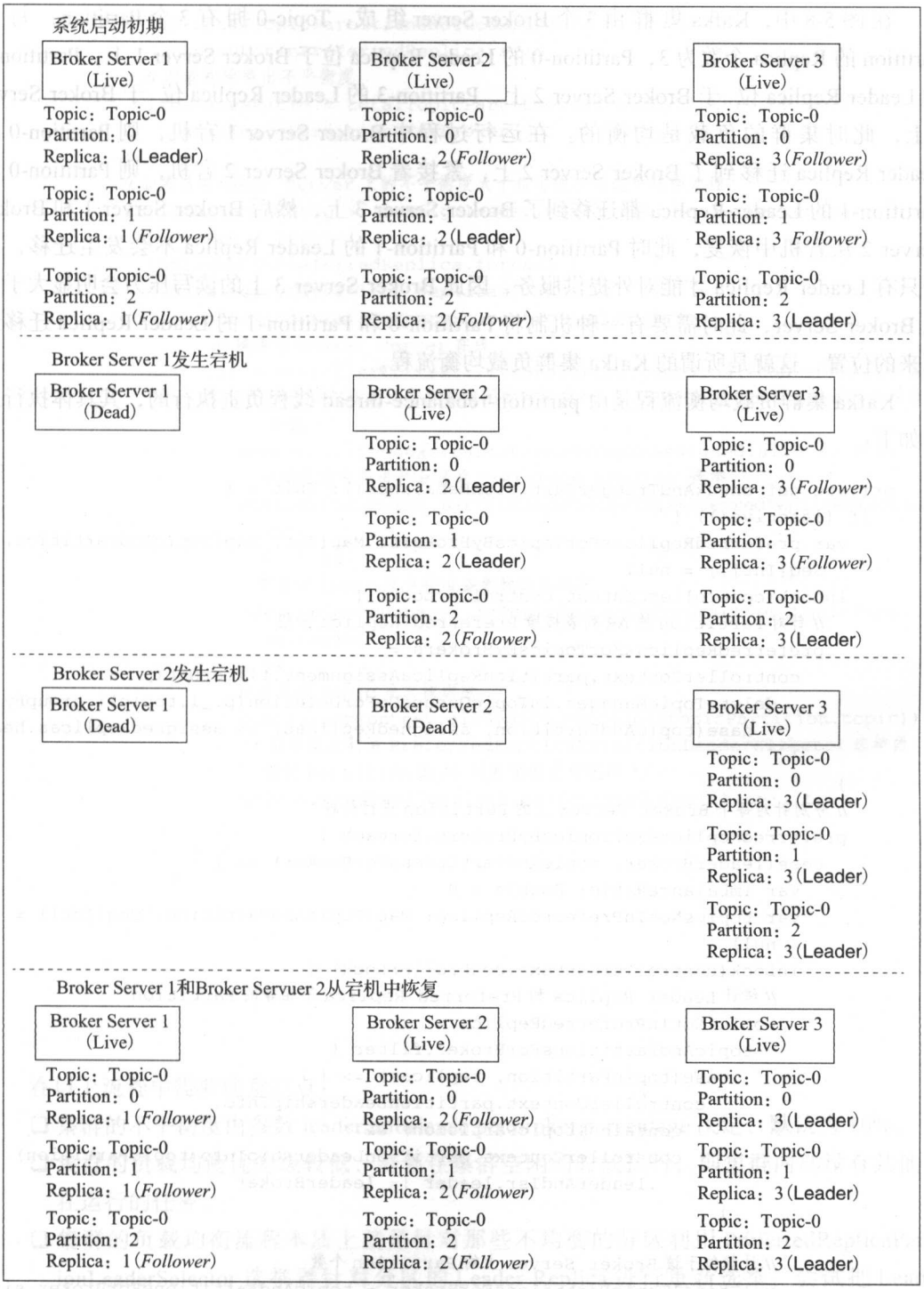


图 5-8　Kafka 集群负载失衡过程

在图 5-8 中，Kafka 集群由 3 个 Broker Server 组成，Topic-0 拥有 3 个 Partition，每个 Partition 的 Replica 个数为 3，Partition-0 的 Leader Replica 位于 Broker Server 1 上，Partition-1 的 Leader Replica 位 于 Broker Server 2 上，Partition-3 的 Leader Replica 位 于 Broker Server 3 上，此时集群的负载是均衡的。在运行过程中 Broker Server 1 宕机，则 Partition-0 的 Leader Replica 迁移到了 Broker Server 2 上，紧接着 Broker Server 2 宕机，则 Partition-0 和 Partition-1 的 Leader Replica 都迁移到了 Broker Server 3 上，然后 Broker Server 1 和 Broker Server 2 从宕机中恢复，此时 Partition-0 和 Partition-1 的 Leader Replica 不会发生迁移，由于只有 Leader Replica 才能对外提供服务，因此 Broker Server 3 上的读写压力会明显大于其他 Broker Server，此时需要有一种机制将 Partition-0 和 Partition-1 的 Leader Replica 迁移到原来的位置，这就是所谓的 Kafka 集群负载均衡流程。

Kafka 集群负载均衡流程是由 partition-rebalance-thread 线程负责执行的，其具体执行逻辑如下：

```
private def checkAndTriggerPartitionRebalance(): Unit = {
  if (isActive()) {
    var preferredReplicasForTopicsByBrokers: Map[Int, Map[TopicAndPartition,
      Seq[Int]]] = null
    inLock(controllerContext.controllerLock) {
      // 针对 Partition 的 AR 列表按照 Preferred Replica 分组
      preferredReplicasForTopicsByBrokers =
        controllerContext.partitionReplicaAssignment.filterNot(p =>
          deleteTopicManager.isTopicQueuedUpForDeletion(p._1.topic)).groupBy {
            case(topicAndPartition, assignedReplicas) => assignedReplicas.head
        }
    }
    // 分别针对每个 Broker Server 上的 Partition 进行处理
    preferredReplicasForTopicsByBrokers.foreach {
      case(leaderBroker, topicAndPartitionsForBroker) => {
        var imbalanceRatio: Double = 0
        var topicsNotInPreferredReplica: Map[TopicAndPartition, Seq[Int]] =
          null
        inLock(controllerContext.controllerLock) {
          // 统计 Leader Replica 和 Preferred Replica 不相等的 Partition
          topicsNotInPreferredReplica =
            topicAndPartitionsForBroker.filter {
              case(topicPartition, replicas) => {
                controllerContext.partitionLeadershipInfo.
                  contains(topicPartition) &&
                  controllerContext.partitionLeadershipInfo(topicPartition)
                    .leaderAndIsr.leader != leaderBroker
              }
            }
          // 计算位于该 Broker Server 上的 Partition 个数
          val totalTopicPartitionsForBroker = topicAndPartitionsForBroker.size
          // 计算 Leader Replica 和 Preferred Replica 不相等的 Partition 的个数
```

```
            val totalTopicPartitionsNotLedByBroker =
              topicsNotInPreferredReplica.size
            // 两者相除得出不平衡度
            imbalanceRatio = totalTopicPartitionsNotLedByBroker.toDouble /
              totalTopicPartitionsForBroker
          }
          // 如果该 Broker Server 上的不平衡度大于指定值，则进行均衡处理
          if (imbalanceRatio > (config.leaderImbalancePerBrokerPercentage.
            toDouble / 100)) {
            topicsNotInPreferredReplica.foreach {
              case(topicPartition, replicas) => {
                inLock(controllerContext.controllerLock) {
                  // 确保该 Broker Server 在线
                  if (controllerContext.liveBrokerIds.contains(leaderBroker) &&
                    /* 由于负载均衡只在集群空闲的时候执行，因此此时系统必须没有在进行分区重
                      分配 */
                    controllerContext.partitionsBeingReassigned.size == 0 &&
                      // 同时没有在进行分区的 Preferred Replica 选举
                    controllerContext.partitionsUndergoingPreferredReplicaElection.
                      size
                      == 0 &&
                    // 并且该 Topic 没有在准备删除的队列中
                    !deleteTopicManager.isTopicQueuedUpForDeletion(topicPartiti
                      on.topic)
                      &&
                    // 并且该 Topic 没有被删除
                    controllerContext.allTopics.contains(topicPartition.topic)) {
                    /* 最后就是利用 PreferredReplicaPartitionLeaderSelector 选举器
                      针对 Partition 的 AR 列表重新进行选举 */
                    onPreferredReplicaElection(Set(topicPartition), true)
                  }
                }
              }
            }
          }
        }
      }
    }
  }
}
```

在以上流程中需要注意三点：

- 集群的不平衡度由参数 leader.imbalance.per.broker.percentage 决定，默认为 10%。
- 集群的负载均衡优先级较低，需要在集群空闲的时候执行，即集群内部没有其他正在运行的任务。
- 集群的负载均衡流程本质上就是针对那些不均衡的分区利用 PreferredReplicaPartitionLeaderSelector 选举器针对分区的 Leader Replica 进行重新选举，尝试把 Leader Replica 切换至 Preferred Replica。

5.8 Kafka 集群的 Topic 删除流程

在 5.6.7 小节提到，当路径为 /admin/delete_topics 的 Zookeeper 上发生数据变化时，此时会触发 DeleteTopicsListener 监听器的 handleChildChange 回调函数，该函数会触发 Topic 的删除，即把 Topic 放入 TopicDeletionManager 的待删除集合，因此真正负责具体 Topic 删除的任务是由 TopicDeletionManager 模块完成的，可见删除 Topic 是一个异步的过程。

Topic 是由 Partition 组成的，而 Partition 是由 Replica 组成的，因此只有 Partition 的 Assigned Replica 全部被删除了，则该 Partition 才可以被删除；只有 Topic 的所有 Partition 都被删除了，则该 Topic 才可以最终被真正地删除。因此关键是如何删除 Replica。回想在 4.3.5.6 小节提到的 StopReplicaRequest，当 Broker Server 接收到该请求时，如果其 deletePartitions=true，则会把 Replica 从该 Broker Server 上删除，因此 TopicDeletionManager 也正是利用这个请求来实现 Replica 的删除。

TopicDeletionManager 内部负责删除 Topic 的线程为 DeleteTopicsThread，为了实现快速删除 Topic，该线程发送 StopReplicaRequest 请求时不会阻塞等待该请求的响应，而是利用 callback 来实现 Replica 状态的切换，如果 StopReplicaResponse 中包含成功删除的 Replica，则将 Replica 的状态从 ReplicaDeletionStarted 切换为 ReplicaDeletionSuccessful。DeleteTopicsThread 不断地检查属于某个 Topic 的所有 Replica 状态，如果 Replica 状态都切换为 ReplicaDeletionSuccessful 的话，则 Topic 可以被删除了。DeleteTopicsThread 线程的执行逻辑如图 5-9 所示。

其中 RequestSendThread 为发送 StopReplicaRequest 的线程。

DeleteTopicsThread 线程对应的大致代码如下：

```
class DeleteTopicsThread() extends ShutdownableThread(……) {
  override def doWork() {
    //等待被触发执行删除
    awaitTopicDeletionNotification()
    if (!isRunning.get)
      return
    inLock(controllerContext.controllerLock) {
      //获取待删除的 Topic 列表
      val topicsQueuedForDeletion = Set.empty[String] ++ topicsToBeDeleted
      topicsQueuedForDeletion.foreach { topic =>
        //Topic 的所有 Replica 都已经被成功删除
        if(controller.replicaStateMachine.areAllReplicasForTopicDeleted(top
          ic)) {
          //彻底删除该 Topic
          completeDeleteTopic(topic)
        } else {
          /*Topic 的所有 Replica 中至少有一个正在开始删除，则不做任何事情，继续等待所有的
            Replica 被删除完成 */
          if(controller.replicaStateMachine.isAtLeastOneReplicaInDeletion
            StartedState(topic)) {
```

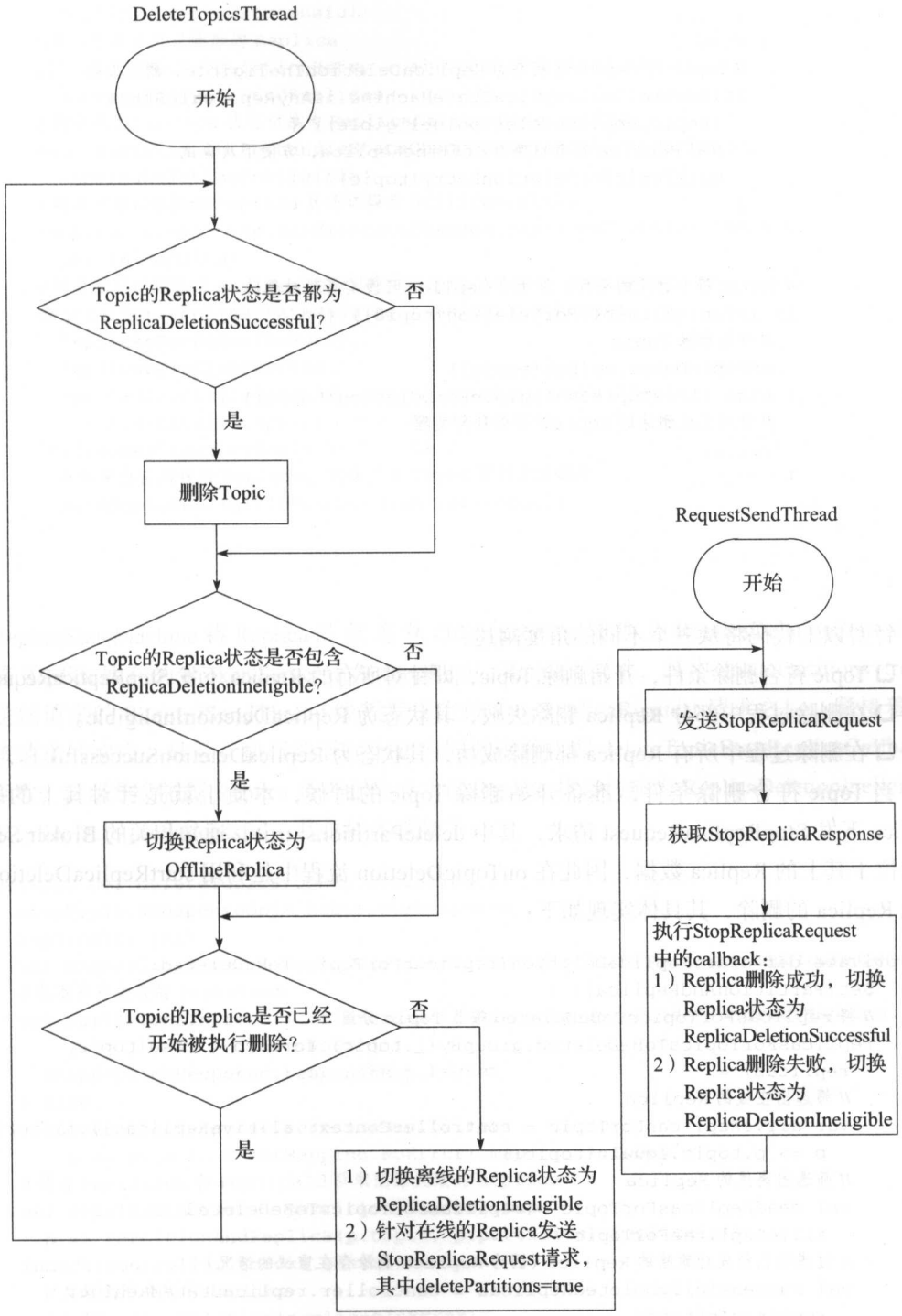


图 5-9　DeleteTopicsThread 线程的执行逻辑

```
            ......
          } else {
            // Topic 的 Replica 状态为 ReplicaDeletionIneligible，删除失败
            if(controller.replicaStateMachine.isAnyReplicaInState
              (topic,ReplicaDeletionIneligible)) {
              // 将 Replica 状态切换为 OfflineReplica，方便下次重试
              markTopicForDeletionRetry(topic)
            }
          }
        }
        // Topic 符合删除的条件，且所有 Replica 还没有开始被删除
        if(isTopicEligibleForDeletion(topic)) {
          // 开始删除 Topic
          onTopicDeletion(Set(topic))
        } else if(isTopicIneligibleForDeletion(topic)) {
          // 针对无法删除的 Topic，不做任何处理
          ......
        }
      }
    }
  }
}
```

针对以上代码将从三个不同的角度阐述：

- Topic 符合删除条件，开始删除 Topic，即针对所有的 Replica 下发 StopReplicaRequest。
- 在删除过程中部分 Replica 删除失败，其状态为 ReplicaDeletionIneligible。
- 在删除过程中所有 Replica 都删除成功，其状态为 ReplicaDeletionSuccessful。

当 Topic 符合删除条件，准备开始删除 Topic 的时候，本质上就是针对其上的所有 Replica 下发 StopReplicaRequest 请求，其中 deletePartitions=true，通知相关的 Broker Server 删除位于其上的 Replica 数据，因此在 onTopicDeletion 流程中是利用 startReplicaDeletion 来实现 Replica 的删除，其具体实现如下：

```
private def startReplicaDeletion(replicasForTopicsToBeDeleted:
  Set[PartitionAndReplica]) {
  // 将 replicasForTopicsToBeDeleted 按照 Topic 分组
  replicasForTopicsToBeDeleted.groupBy(_.topic).foreach { case(topic,
    replicas) =>
    // 筛选出在线的 Replica
    var aliveReplicasForTopic = controllerContext.allLiveReplicas().filter(
      p => p.topic.equals(topic))
    // 筛选出离线的 Replica
    val deadReplicasForTopic = replicasForTopicsToBeDeleted -
      aliveReplicasForTopic
    // 过滤出已经成功删除的 Replica（因为 Replica 删除存在重试的情况）
    val successfullyDeletedReplicas = controller.replicaStateMachine.
      replicasInState(
      topic,
```

```
        ReplicaDeletionSuccessful)
      // 筛选出真正可以删除的 Replica
      val replicasForDeletionRetry = aliveReplicasForTopic -
        successfullyDeletedReplicas
      // 将离线的 Replica 状态切换为 ReplicaDeletionIneligible
      replicaStateMachine.handleStateChanges(deadReplicasForTopic,
        ReplicaDeletionIneligible)
      // 将真正可以删除的 Replica 状态切换为 OfflineReplica
      replicaStateMachine.handleStateChanges(replicasForDeletionRetry,
        OfflineReplica)
      // 将真正可以删除的 Replica 状态切换为 ReplicaDeletionStarted
      controller.replicaStateMachine.handleStateChanges(
        replicasForDeletionRetry,
        ReplicaDeletionStarted,
        new Callbacks.CallbackBuilder().stopReplicaCallback
          (deleteTopicStopReplicaCallback).build)
      if(deadReplicasForTopic.size > 0) {
        // 如果存在离线的 Replica，则标记该 Topic 暂时无法删除
        markTopicIneligibleForDeletion(Set(topic))
      }
    }
  }
```

ReplicaStateMachine 将 Replica 的状态从 OfflineReplica 转换为 ReplicaDeletionStarted 时会向该 Replica 所在的 Broker Server 下发 StopReplicaRequest 请求，其中 deletePartitions=true，当发送该请求的 RequestSendThread 线程发送完该 StopReplicaRequest 请求之后，紧接着会等待该请求的响应，最后会执行注册在上面的回调函数，即 deleteTopicStopReplicaCallback 回调函数，该函数会根据 Replica 是否删除成功将 Replica 状态切换为 ReplicaDeletionIneligible 或者 ReplicaDeletionSuccessful，其具体实现如下：

```
  private def deleteTopicStopReplicaCallback(
    stopReplicaResponseObj: RequestOrResponse,
    replicaId: Int) {
    val stopReplicaResponse = stopReplicaResponseObj.asInstanceOf[StopReplicaResponse]
    // 获取删除失败的 Partition
    val partitionsInError = if(stopReplicaResponse.errorCode != ErrorMapping.
                               NoError) {
      stopReplicaResponse.responseMap.keySet
    } else
      stopReplicaResponse.responseMap.filter(
        p => p._2 != ErrorMapping.NoError).map(_._1).toSet
    // 通过 Partition 和 replicaId 获取删除失败的 Replica
    val replicasInError = partitionsInError.map(
      p => PartitionAndReplica(p.topic, p.partition, replicaId))
    inLock(controllerContext.controllerLock) {
      // 处理删除失败的 Replica
      failReplicaDeletion(replicasInError)
      if(replicasInError.size != stopReplicaResponse.responseMap.size) {
```

```
        // 获取删除成功的 Replica
        val deletedReplicas = stopReplicaResponse.responseMap.keySet --
          partitionsInError
        // 处理删除成功的 Replica
        completeReplicaDeletion(deletedReplicas.map(
          p => PartitionAndReplica(p.topic, p.partition, replicaId)))
      }
    }
  }
```

如果 Broker Server 正在处于 Shutdown 流程中，或者 Partition 正在处于 Reassignment 流程中，或者 Partition 正在处于 Preferred Replica Election 中，此时 Replica 会删除失败，那么针对删除失败的 Replica，除了将其状态切换为 ReplicaDeletionIneligible，还需要标记该 Topic 暂时无法删除，failReplicaDeletion 的具体实现如下：

```
def failReplicaDeletion(replicas: Set[PartitionAndReplica]) {
  if(isDeleteTopicEnabled) {
    // 过滤出删除失败的 Replica
    val replicasThatFailedToDelete = replicas.filter(r =>
      isTopicQueuedUpForDeletion(r.topic))
    if(replicasThatFailedToDelete.size > 0) {
      val topics = replicasThatFailedToDelete.map(_.topic)
      // 将 Replica 的状态切换为 ReplicaDeletionIneligible
      controller.replicaStateMachine.handleStateChanges(
        replicasThatFailedToDelete,
        ReplicaDeletionIneligible)
      // 标记该 Topic 暂时无法删除
      markTopicIneligibleForDeletion(topics)
      // 触发 DeleteTopicsThread 线程执行
      resumeTopicDeletionThread()
    }
  }
}
```

如果 Replica 删除成功，则将其状态转换为 ReplicaDeletionSuccessful，completeReplicaDeletion 的具体实现如下：

```
private def completeReplicaDeletion(replicas: Set[PartitionAndReplica]) {
  // 过滤出删除成功的 Replica
  val successfullyDeletedReplicas = replicas.filter(r =>
    isTopicQueuedUpForDeletion(r.topic))
  // 将 Replica 的状态切换为 ReplicaDeletionSuccessful
  controller.replicaStateMachine.handleStateChanges(
    successfullyDeletedReplicas,
    ReplicaDeletionSuccessful)
  // 触发 DeleteTopicsThread 线程执行
  resumeTopicDeletionThread()
}
```

当Topic删除过程中出现部分Replica删除失败的情况时，为了再次触发删除，需要将Replica的状态从ReplicaDeletionIneligible转换为OfflineReplica，因此markTopicForDeletionRetry就是将Replica状态转换为OfflineReplica，其具体实现如下：

```
private def markTopicForDeletionRetry(topic: String) {
  // 获取删除失败的 Replica
  val failedReplicas = controller.replicaStateMachine.replicasInState(
    topic,
    ReplicaDeletionIneligible)
  // 将其状态切换为 OfflineReplica
  controller.replicaStateMachine.handleStateChanges(
    failedReplicas,
    OfflineReplica)
}
```

当Topic的所有Replica都被删除的时候（Replica的状态都为ReplicaDeletionSuccessful），则首先会将Replica的状态从ReplicaDeletionSuccessful转换为NonExistentReplica，然后将Partition的状态一步一步从原始状态转换为NonExistentPartition状态，中间经历OfflinePartition状态，接着从Zookeeper上删除该Topic相关的数据，最后从ControllerContext内部移除该Topic相关的数据，整体流程如下：

```
private def completeDeleteTopic(topic: String) {
  // 注销针对 /brokers/topics/[topic] 目录的监听
  partitionStateMachine.deregisterPartitionChangeListener(topic)
  // 获取该 Topic 对应的 Replica
  val replicasForDeletedTopic = controller.replicaStateMachine.
    replicasInState(topic, ReplicaDeletionSuccessful)
  // 将 Replica 状态切换为 NonExistentReplica
  replicaStateMachine.handleStateChanges(
    replicasForDeletedTopic,
    NonExistentReplica)
  // 获取该 Topic 对应的 Partition
  val partitionsForDeletedTopic = controllerContext.partitionsForTopic(topic)
  // 切换 Partition 状态为 OfflinePartition
  partitionStateMachine.handleStateChanges(
    partitionsForDeletedTopic,
    OfflinePartition)
  // 切换 Partition 状态为 NonExistentPartition
  partitionStateMachine.handleStateChanges(
    partitionsForDeletedTopic,
    NonExistentPartition)
  // 从待删除集合中剔除
  topicsToBeDeleted -= topic
  partitionsToBeDeleted.retain(_.topic != topic)
  // 清理 Zookeeper 上相关的数据
  controllerContext.zkClient.deleteRecursive(ZkUtils.getTopicPath(topic))
  controllerContext.zkClient.deleteRecursive(ZkUtils.
    getTopicConfigPath(topic))
```

```
        controllerContext.zkClient.delete(ZkUtils.getDeleteTopicPath(topic))
        // 从 ControllerContext 内部移除该 Topic 相关的数据
        controllerContext.removeTopic(topic)
    }
```

总而言之，Topic 的删除是一个完全异步的流程。Leader 状态的 KafkaController 需要不断地收集每个 Replica 的删除状态，只有当 Topic 的所有 Replica 都被成功删除，Topic 才可以被彻底删除，只要 Topic 的某个 Replica 删除失败，Topic 就无法被删除。

5.9 KafkaController 的通信模块

ControllerChannelManager 提供了 Leader 状态的 KafkaController 和集群其他 Broker Server 通信的功能，内部针对每一个在线的 Broker Server 会维护一个通信链路，并分别通过各自的 RequestSendThread 线程将请求发送给对应的 Broker Server。ControllerBrokerRequestBatch 内部缓存了以上提到的请求，分别为：LeaderAndIsrRequest、StopReplicaRequest 和 UpdateMetadataRequest，通过将请求按照接收端（Broker Server）的不同而分发到各自 RequestSendThread 内部的阻塞队列中，RequestSendThread 从各自的阻塞队列中获取相应的请求并转发到对应的 Broker Server，其相互之间的关系如图 5-10 所示。

接下来将从三个方面讲解以上的流程：

- ControllerChannelManager 和 Broker Server 通信链路的建立时机。
- BlockingChannel 的具体实现。
- RequestSendThread 的发送流程。

ControllerChannelManager 和 Broker Server 通信链路的建立时机主要发生在两个时刻：

1）ControllerChannelManager 初始化的时候，会针对当前在线的 Broker Server 分别与之建立通信链路。

2）由于在 /brokers/ids 的 Zookeeper 路径上注册了 BrokerChangeListener 监听器，当发现有新上线的 Broker Server 时，则也会与其建立通信链路。

RequestSendThread 内部和 Broker Server 的通信链路本质上是通过 Socket 通信来实现的，RequestSendThread 内部的 channel 是 Socket 客户端，Broker Server 是 Socket 服务端，其具体实现如下：

```
class BlockingChannel(val host: String,
                      val port: Int,
                      val readBufferSize: Int,
                      val writeBufferSize: Int,
                      val readTimeoutMs: Int ) extends Logging {
  private var connected = false
  // Socket 客户端
```

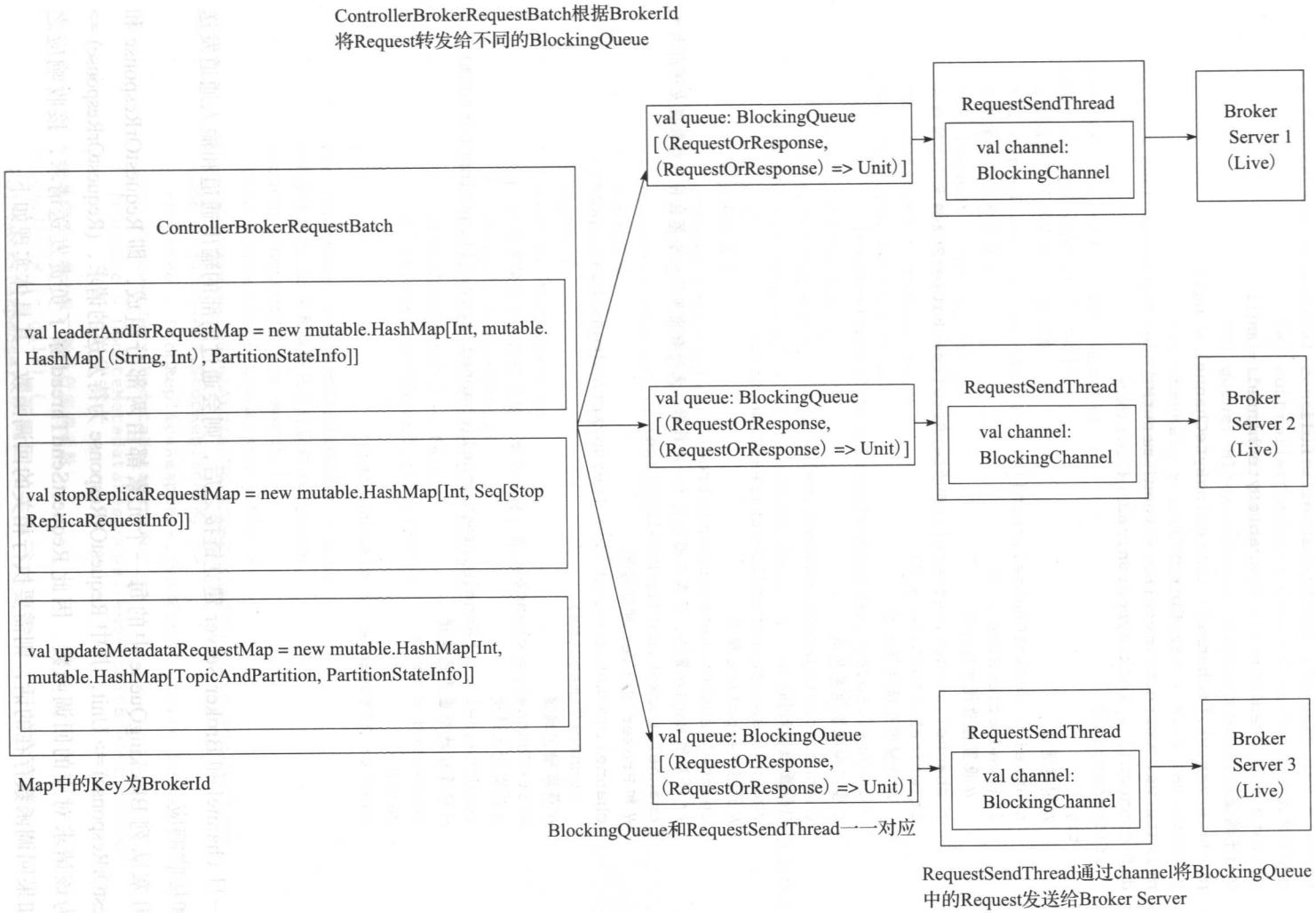


图 5-10　KafkaController 通信模块内部原理

```
    private var channel: SocketChannel = null
    //输入通道
    private var readChannel: ReadableByteChannel = null
    //输出通道
    private var writeChannel: GatheringByteChannel = null
    private val lock = new Object()
    private val connectTimeoutMs = readTimeoutMs
    def connect() = lock synchronized  {
      if(!connected) {
        try {
          //创建通道
          channel = SocketChannel.open()
          if(readBufferSize > 0)
            //设置接收缓冲区大小
            channel.socket.setReceiveBufferSize(readBufferSize)
          if(writeBufferSize > 0)
            //设置发送缓冲区大小
            channel.socket.setSendBufferSize(writeBufferSize)
          //客户端配置阻塞模式
          channel.configureBlocking(true)
          //设置超时时间
          channel.socket.setSoTimeout(readTimeoutMs)
          //开启 KeepAlive 模式
          channel.socket.setKeepAlive(true)
          /* 不启用 Nagle 算法，即客户端每发送一次数据，无论数据包的大小都会将这些数据发送出去 */
          channel.socket.setTcpNoDelay(true)
          //和 Broker Server 建立链接
          channel.socket.connect(new InetSocketAddress(host, port),
            connectTimeoutMs)
          //获取输出通道
          writeChannel = channel
          //获取输入通道
          readChannel = Channels.newChannel(channel.socket().getInputStream)
          //将连接状态置为已连接
          connected = true
        } catch {
          case e: Throwable => disconnect()
        }
      }
    }
}
```

一旦 channel 和 Broker Server 建立链接之后，则会通过内部的输出通道和输入通道发送请求和接收响应。

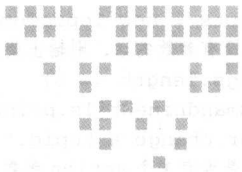
阻塞队列 BlockingQueue 中的每一个元素都由两部分组成，即 RequestOrResponse 和 (RequestOrResponse) => Unit，其中 RequestOrResponse 为待发送的请求，(RequestOrResponse) => Unit 为该请求对应的回调函数。因此 RequestSendThread 除了负责发送请求，接收响应之外，如果回调函数存在的话，则需要执行相关的回调函数，其具体实现如下：

```
class RequestSendThread(val controllerId: Int,
                        val controllerContext: ControllerContext,
                        val toBroker: Broker,
                        val queue: BlockingQueue[(RequestOrResponse,
                                                  (RequestOrResponse) => Unit)],
                        val channel: BlockingChannel)
  extends ShutdownableThread("Controller-%d-to-broker-%d-send-thread".format(
                             controllerId,toBroker.id)) {
  private val lock = new Object()
  override def doWork(): Unit = {
    //从阻塞队列中获取二元组对象
    val queueItem = queue.take()
    //取第一个元素为请求
    val request = queueItem._1
    //取第二个元素为回调函数
    val callback = queueItem._2
    var receive: Receive = null
    try {
      lock synchronized {
        var isSendSuccessful = false
        while(isRunning.get() && !isSendSuccessful) {//重试机制，直到发送成功为止
          try {
            //发送请求
            channel.send(request)
            //获取响应（二进制数据）
            receive = channel.receive()
            //发送成功
            isSendSuccessful = true
          } catch {
            case e: Throwable =>
              //异常情况下，断开连接，重新连接，等待片刻之后再次发送
              channel.disconnect()
              connectToBroker(toBroker, channel)
              isSendSuccessful = false
              Utils.swallow(Thread.sleep(300))
          }
        }
        var response: RequestOrResponse = null
        //将接收到的二进制数据转化为对应的response
        request.requestId.get match {
          case RequestKeys.LeaderAndIsrKey =>
            response = LeaderAndIsrResponse.readFrom(receive.buffer)
          case RequestKeys.StopReplicaKey =>
            response = StopReplicaResponse.readFrom(receive.buffer)
          case RequestKeys.UpdateMetadataKey =>
            response = UpdateMetadataResponse.readFrom(receive.buffer)
        }
        //如果存在回调函数，则执行回调函数
        if(callback != null) {
          callback(response)
```

```
            }
          }
        } catch {
          case e: Throwable =>
            channel.disconnect()
        }
      }
    }
```

5.10 本章小结

KafkaController 模块作为整个 Kafka 集群的控制管理者，在若干个 Broker 内部有且只有一个对外提供服务，并且 KafkaController 模块利用 Zookeeper 模块对外提供高可靠性。理解分区的状态切换，副本的状态切换以及 KafkaController 内部的监听器是理解整个 KafkaController 模块的关键。至此有关 Broker 内部的基本模块和控制管理模块都已经讲解完毕，在后续章节中将主要描述如何使用 Kafka 消息系统。



第 6 章 Chapter 6

Topic 的管理工具

本章主要讲解 Topic 管理工具的实现原理。从第 4 章可以看出，Broker Server 并没有直接提供管理 Topic 的二进制协议，而是通过监测 Zookeeper 上相关路径数据的变化来间接触发 Topic 的管理，因此有关 Topic 的管理工具本质上是通过修改、删除或者增加 Zookeeper 上相关路径上的数据来触发 Leader 状态的 KafkaController 管理 Topic，因此 Topic 的管理都是一个异步的流程。

Kafka 对外提供了若干个维护脚本，其中 kafka-topics.sh 涉及 Topic 的创建、修改、列举、描述和删除，kafka-reassign-partitions.sh 涉及分区的重分配，kafka-preferred-replica-election.sh 涉及分区副本 Leader 的选举，本章将着重描述以上三个工具的使用方法、实现原理和注意事项。

6.1 kafka-topics.sh

kafka-topics.sh 提供了 Topic 的创建、修改、列举、描述、删除功能，在内部是通过 TopicCommand 类来实现的，其脚本内容如下：

```
// kafka-run-class.sh 加载 kafka 的 classpath，执行其中的 kafka.admin.TopicCommand
// 类的 main 函数
exec $(dirname $0)/kafka-run-class.sh kafka.admin.TopicCommand $@
```

通过解析不同的 action 参数来执行不同的流程，如下所示：

```
object TopicCommand {
  def main(args: Array[String]): Unit = {
```

```
    //解析参数
    val opts = new TopicCommandOptions(args)
    //如果没有参数的话，则输出使用方法并退出
    if(args.length == 0)
      CommandLineUtils.printUsageAndDie(opts.parser, "Create, delete, describe,
        or change a topic.")
    //判断是否有多个 action 参数
    val actions = Seq(opts.createOpt,
                      opts.listOpt,
                      opts.alterOpt,
                      opts.describeOpt,
                      opts.deleteOpt).count(opts.options.has _)
    //一次只支持一种 action，否则命令无效
    if(actions != 1)
      CommandLineUtils.printUsageAndDie(
        opts.parser, "Command must include exactly one action: --list, --describe,
          --create, --alter or --delete")
    //校验参数的有效性
    opts.checkArgs()
    //创建 Zookeeper 连接
    val zkClient = new ZkClient(opts.options.valueOf(opts.zkConnectOpt),
                                30000,
                                30000,
                                ZKStringSerializer)
    try{
      if(opts.options.has(opts.createOpt))
        //包含 create 关键字，则创建 topic
        createTopic(zkClient, opts)
      else if(opts.options.has(opts.alterOpt))
        //包含 alter 关键字，则修改 topic
        alterTopic(zkClient, opts)
      else if(opts.options.has(opts.listOpt))
        //包含 list 关键字，则列举 topic
        listTopics(zkClient, opts)
      else if(opts.options.has(opts.describeOpt))
        //包含 describe 关键字，则描述 topic
        describeTopic(zkClient, opts)
      else if(opts.options.has(opts.deleteOpt))
        //包含 delete 关键字，则删除 topic
        deleteTopic(zkClient, opts)
    }catch{
      case e: Throwable =>
} finally {
      //最后关闭 Zookeeper 连接
      zkClient.close()
    }
  }
}
```

其中支持以下若干个 action 参数：

- --create 创建 Topic。
- --alter 修改 Topic 配置。
- --list 列举 Topic。
- --describe 描述 Topic。
- --delete 删除 Topic。

通过以上五种不同的 action 类型，可以实现 Topic 生命周期的基本管理，接下来将详细描述以上 action 的控制台输出和内部详细的执行原理。

6.1.1　createTopic

在 Kafka 安装包的 bin 目录下执行图 6-1 所示命令来创建 Topic。

```
[root@ localhost bin]# pwd
/usr/dahua/kafka/bin
[root@ localhost bin]# ./kafka-topics.sh --create --zookeeper
  localhost:2181 --replication-factor 2 --partitions 2 --topic test
Created topic "test".
```

图 6-1　创建 Topic

其中：

- --create 设置此次操作的 action 类型为创建。
- --zookeeper localhost:2181 设置 Zookeeper 集群的地址。
- --replication-factor 设置 Topic 的副本因子。
- --partitions 设置 topic 的分区个数。
- --topic 设置 topic 的名称。

此时由于没有指定 Partition 的 AR 列表，则会根据负载均衡算法将 Partition 的 Replica 均衡地分布在 Kafka 集群中，并且 Leader Replica 为 AR 列表中的第一个 Replica，即 Prefered Replica。如果指定了类似于 --replica-assignment 1:2,2:3,3:1 的参数，则设置了 Partition 的 AR 列表，其中以逗号区别不同的 Partition，以冒号区别相同 Partition 的 AR 列表中的不同 Replica，即 Partition 0 的 AR=[1,2]，Partition 1 的 AR=[2,3] 和 Partition 2 的 AR=[3,1]。createTopic 的流程如下：

```
def createTopic(zkClient: ZkClient, opts: TopicCommandOptions) {
  //解析出 topic 名称
  val topic = opts.options.valueOf(opts.topicOpt)
  //解析出配置文件
  val configs = parseTopicConfigsToBeAdded(opts)
  if (opts.options.has(opts.replicaAssignmentOpt)) {
    //包含 --replica-assignment，则提取其中的具体参数
    val assignment = parseReplicaAssignment(opts.options.valueOf(opts.
      replicaAssignmentOpt))
    /* 将 topic 的配置参数和 replica 的分布情况分别持久化至 /config/topics/[topic] 目录和
       "/brokers/topics/[topic] 目录 */
```

```
        AdminUtils.createOrUpdateTopicPartitionAssignmentPathInZK(zkClient, topic,
          assignment, configs)
      } else {
        /* 不包含 --replica-assignment，则需要自动进行分配，因此输入参数中必须包含—partitions
          和 --replication-factor*/
        CommandLineUtils.checkRequiredArgs(
          opts.parser,
          opts.options,
          opts.partitionsOpt,
          opts.replicationFactorOpt)
        // 提取 --partitions 的具体值
        val partitions = opts.options.valueOf(opts.partitionsOpt).intValue
        // 提取 --replication-factor 的具体值
        val replicas = opts.options.valueOf(opts.replicationFactorOpt).intValue
        // 开始创建 Topic
        AdminUtils.createTopic(zkClient, topic, partitions, replicas, configs)
      }
      println("Created topic \"%s\".".format(topic))
    }
```

一般情况下用户不会指定 --replica-assignment 参数，而是由 Kafka 提供的默认分配算法决定。Kafka 默认的分配算法有 2 个主要的原则：

- 针对 Topic 所有的 Replicas，需要将其均匀地分布在所有的 Broker Server 上。
- 针对 Partition 内的 Replicas，需要将其均匀地分布在不同的 Broker Server 上。

因此分配算法的主要流程如下：

1）从 Broker Server 的随机位置开始按照轮询的方式选择每个 Partiton 的 First Replica。

2）不同 Partition 剩余的 Replica 按照一定的偏移量紧跟着各自 Partition 的 First Replica。

假设当前 Kafka 集群由 3 个 Broker Server 组成，分别为 Broker-0、Broker-1 和 Broker-2，此时准备创建分区个数为 3，副本因子为 2 的 Topic，其中 P0-1 代表分区 0 的第一个副本，其他类推，则分配流程如图 6-2 所示。

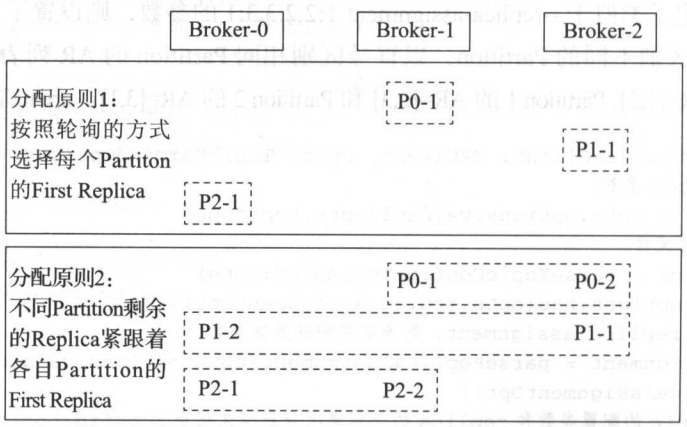


图 6-2 Topic 的 Replica Assignment 流程

其中 P0-1 正好随机到了 Broker-1 之上，之后 P1-1 和 P2-1 按照轮询的方式分配，并且每个分区的第二个 Replica 距离第一个 Replica 的长度正好为一个分区。AdminUtils.createTopic 的具体实现如下：

```
def createTopic(zkClient: ZkClient,
                topic: String,
                partitions: Int,
                replicationFactor: Int,
                topicConfig: Properties = new Properties) {
  //在分配之前需要将 Broker 按照 id 排序
  val brokerList = ZkUtils.getSortedBrokerList(zkClient)
  //执行分配算法
  val replicaAssignment = AdminUtils.assignReplicasToBrokers(brokerList, partitions,
    replicationFactor)
  /* 将 topic 的配置参数和 replica 的分布情况分别持久化至 /config/topics/[topic] 目录和
     "/brokers/topics/[topic] 目录 */
  AdminUtils.createOrUpdateTopicPartitionAssignmentPathInZK(
    zkClient,
    topic,
    replicaAssignment,
    topicConfig)
}
```

其中 assignReplicasToBrokers 方法负责具体的分配算法，具体实现代码如下：

```
def assignReplicasToBrokers(brokerList: Seq[Int],
                            nPartitions: Int,
                            replicationFactor: Int,
                            fixedStartIndex: Int = -1,
                            startPartitionId: Int = -1): Map[Int, Seq[Int]] = {
  //nPartitions 参数必须大于 0
  if (nPartitions <= 0)
    throw new AdminOperationException("number of partitions must be larger than 0")
  //replicationFactor 参数必须大于 0
  if (replicationFactor <= 0)
    throw new AdminOperationException("replication factor must be larger than 0")
  /*replicationFactor 不能大于 Broker 个数，否则同一个 Broker 上将分配到相同 Partition 的
    不同 Replica*/
  if(replicationFactor > brokerList.size)
    throw new AdminOperationException("replication factor: " + replicationFactor +
      " larger than available brokers: " + brokerList.size)
  val ret = new mutable.HashMap[Int, List[Int]]()
  //起始偏移量，如果小于 0 的话，则取随机数
  val startIndex = if (fixedStartIndex >= 0) fixedStartIndex else rand.nextInt
    (brokerList.size)
  //起始分配的 Partition，默认从第一个 Partition 开始
  var currentPartitionId = if (startPartitionId >= 0) startPartitionId else 0
  //计算相同分区内第二个 Replica 距离第一个 Replica 的偏移量
  var nextReplicaShift = if (fixedStartIndex >= 0) fixedStartIndex else rand.
    nextInt(brokerList.size)
```

```
    for (i <- 0 until nPartitions) {
      // 如果 nextReplicaShift 正好等于 Broker 个数，则 +1，从第一个 Broker 开始
      if (currentPartitionId > 0 && (currentPartitionId % brokerList.size == 0))
        nextReplicaShift += 1
      // 计算分区的第一个 Replica 索引
      val firstReplicaIndex = (currentPartitionId + startIndex) % brokerList.
        size
      // 获取对应的 Broker Id
      var replicaList = List(brokerList(firstReplicaIndex))
      for (j <- 0 until replicationFactor - 1)
        // 分配分区内其他的 Replica，距离第一个 Replica 的偏移量为 nextReplicaShift
        replicaList ::= brokerList(replicaIndex(firstReplicaIndex, nextReplicaShift,
          j, brokerList.size))
      // 分配结束，保存起来
      ret.put(currentPartitionId, replicaList.reverse)
      // 分区索引 +1，继续分配
      currentPartitionId = currentPartitionId + 1
    }
    ret.toMap
  }
```

无论用户是否传入 --replica-assignment 相关参数，一旦 TopicCommand 将 topic 的配置参数和 replica 的分布情况分别持久化至 /config/topics/[topic] 和 /brokers/topics/[topic] 的 Zookeeper 目录上之后，会触发 5.6.1 小节提到的 TopicChangeListener 监听器，从而触发 Topic 的真正创建。

6.1.2 alterTopic

在 Kafka 安装包的 bin 目录下执行图 6-3 所示的命令来增加 Topic 分区数目。

```
[root@ localhost bin]# pwd
/usr/dahua/kafka/bin
[root@ localhost bin]# ./kafka-topics.sh --zookeeper
  localhost:2181 --alter --topic test --partitions 4
WARNING: If partitions are increased for a topic that has a key,
  the partition logic or ordering of the messages will be affected
Adding partitions succeeded!
```

图 6-3 增加 Topic 分区

其中：

- --alter 设置此次操作的 action 类型为修改。
- --zookeeper localhost:2181 设置 Zookeeper 集群的地址。
- --topic 设置 topic 的名称。
- --partitions 设置 Topic 的分区数目，如果当前没有达到，则增加分区个数。

在 Kafka 安装包的 bin 目录下执行图 6-4 所示的命令来修改 Topic 配置。

```
[root@ localhost bin]# pwd
/usr/dahua/kafka/bin
[root@ localhost bin]# ./kafka-topics.sh --zookeeper
  localhost:2181 --alter --topic test --config flush.messages=1
Updated config for topic "test".
```

图 6-4　修改 Topic 配置

其中：

- --alter 设置此次操作的 action 类型为修改。
- --zookeeper localhost:2181 设置 Zookeeper 集群的地址。
- --topic 设置 topic 的名称。
- --config flush.messages=1 设置需要修改的配置项。

在 Kafka 安装包的 bin 目录下执行图 6-5 所示的命令来删除 Topic 配置。

```
[root@ localhost bin]# pwd
/usr/dahua/kafka/bin
[root@ localhost bin]# ./kafka-topics.sh --zookeeper
  localhost:2181 --alter --topic test --delete-config flush.messages
Updated config for topic "test".
```

图 6-5　删除 Topic 配置

其中：

- --alter 设置此次操作的 action 类型为修改。
- --zookeeper localhost:2181 设置 Zookeeper 集群的地址。
- --topic 设置 topic 的名称。
- --delete-config flush.messages=1 设置需要删除的配置项。

以上三个方法都是通过设置 action 类型为 alter 来实现的，其具体实现如下：

```
def alterTopic(zkClient: ZkClient,
               opts: TopicCommandOptions) {
  val topics = getTopics(zkClient, opts)
  topics.foreach { topic =>
    // 从 /config/topics/[topic] 加载 Topic 的配置参数
    val configs = AdminUtils.fetchTopicConfig(zkClient, topic)
    // 判断是否增加或者删除 config
    if(opts.options.has(opts.configOpt) || opts.options.has(opts.deleteConfigOpt)) {
      // 从 --config 参数后面获取需要修改的配置项
      val configsToBeAdded = parseTopicConfigsToBeAdded(opts)
      // 从 --delete-config 参数后面获取需要删除的配置项
      val configsToBeDeleted = parseTopicConfigsToBeDeleted(opts)
      // 合并待修改的配置项
      configs.putAll(configsToBeAdded)
      // 合并删除的配置项
```

```
        configsToBeDeleted.foreach(config => configs.remove(config))
        // 将 Config 上传至 Zookeeper
        AdminUtils.changeTopicConfig(zkClient, topic, configs)
      }
      // 判断是否增加 partition
      if(opts.options.has(opts.partitionsOpt)) {
        // 不可以增加 Kafka 集群内部保留的名为 __consumer_offsets 的 Topic
        if (topic == OffsetManager.OffsetsTopicName) {
          throw new IllegalArgumentException("The number of partitions for the
            offsets topic cannot be changed.")
        }
        // 获取分区总数
        val nPartitions = opts.options.valueOf(opts.partitionsOpt).intValue
        // 获取 AR 列表
        val replicaAssignmentStr = opts.options.valueOf(opts.replicaAssignmentOpt)
        // 上传 Topic 的配置情况至 Zookeeper
        AdminUtils.addPartitions(zkClient, topic, nPartitions,
          replicaAssignmentStr,config = configs)
      }
    }
  }
```

回想 4.5 小节提到的 Topic 配置文件的修改，首先需要将修改的通知写入 /brokers/config_changes 目录，然后触发 TopicConfigManager 从 /brokers/config_changes 目录读取待修改的 Topic 名称，然后再从 /brokers/topics/[topic]/config 加载配置，这正是 AdminUtils.changeTopicConfig 所要完成的事情，其内部实现如下：

```
def changeTopicConfig(zkClient: ZkClient,
                      topic: String,
                      configs: Properties) {
  // 判断 Topic 是否存在
  if(!topicExists(zkClient, topic))
    throw new AdminOperationException("Topic \"%s\" does not exist.".format(topic))
  // 校验配置项是否正确
  LogConfig.validate(configs)
  // 将 config 写入 /config/topics/[topic] 目录
  writeTopicConfig(zkClient, topic, configs)
  // 将 config change 通知写入 /config/changes 目录
  zkClient.createPersistentSequential(
    ZkUtils.TopicConfigChangesPath + "/" + TopicConfigChangeZnodePrefix,
    Json.encode(topic))
}
```

当 changeTopicConfig 完成以上步骤之后，就会触发 TopicConfigManager 修改 Topic 配置参数的流程，最终完成 Topic 配置参数的修改。

在 6.1.1 小节提到 Kafka 集群 Replica 的默认分配算法，如果此时需要增加 Partition，但是没有传入用户指定的分区 AR 列表，则会按照默认的分配算法分配 replica，其内部实现如下：

```
def addPartitions(zkClient: ZkClient,
                  topic: String,
                  numPartitions: Int = 1,
                  replicaAssignmentStr: String = "",
                  checkBrokerAvailable: Boolean = true,
                  config: Properties = new Properties) {
  // 从 /brokers/topics/[topic] 目录加载 Partition 的 AR 列表
  val existingPartitionsReplicaList = ZkUtils.getReplicaAssignmentForTopics
    (zkClient, List(topic))
  // 判断 Topic 是否存在
  if (existingPartitionsReplicaList.size == 0)
    throw new AdminOperationException("The topic %s does not exist".
      format(topic))
  // 获取第一个 Partition 的 AR 列表
  val existingReplicaList = existingPartitionsReplicaList.head._2
  // 计算需要增加的 Partition 个数
  val partitionsToAdd = numPartitions - existingPartitionsReplicaList.size
  // Partition 只能增加，不能减少
  if (partitionsToAdd <= 0)
    throw new AdminOperationException("The number of partitions for a topic
      can only be increased")
  // 获取排序的 BrokerId 列表
  val brokerList = ZkUtils.getSortedBrokerList(zkClient)
  val newPartitionReplicaList = if (replicaAssignmentStr == null ||
    replicaAssignmentStr == "")
    /* 没有指定 --replica-assignment 参数，或者该参数为空，则采用默认的 Replica 分配算法，
       参考 6.1.1 小节 */
    AdminUtils.assignReplicasToBrokers(
      brokerList,
      partitionsToAdd,
      existingReplicaList.size,
      existingReplicaList.head,
      existingPartitionsReplicaList.size)
  else
    /*--replica-assignment 参数包含 Partition 的 AR 列表，则从 --replica-assignment 中
      加载，其中逗号区分不同的 Partiton，冒号区分相同 Partition 的不同 Replica*/
    getManualReplicaAssignment(
      replicaAssignmentStr,
      brokerList.toSet,
      existingPartitionsReplicaList.size,
      checkBrokerAvailable)
  // 确认新增的 Partition 是否达到 Topic 的副本因子
  val unmatchedRepFactorList = newPartitionReplicaList.values.filter(
    p => (p.size != existingReplicaList.size))
  if (unmatchedRepFactorList.size != 0)
    throw new AdminOperationException("The replication factor in manual
      replication assignment " +
      " is not equal to the existing replication factor for the topic " +
        existingReplicaList.size)
  val partitionReplicaList = existingPartitionsReplicaList.map(p => p._1.
    partition -> p._2)
```

```
    // 合并新增的 partition
    partitionReplicaList ++= newPartitionReplicaList
    // 将 config 和 AR 分别写入 /config/topics/[config] 和 /brokers/topics/[topic] 目录
    AdminUtils.createOrUpdateTopicPartitionAssignmentPathInZK(
      zkClient,
      topic,
      partitionReplicaList,
      config,
      true)
  }
```

当 addPartitions 完成以上步骤之后，就会触发 5.6.2 小节中提到的 AddPartitionsListener 监听器的新增 Partition 的流程，最终完成 Partition 的新增操作。

6.1.3 listTopics

在 Kafka 安装包的 bin 目录下执行图 6-6 所示的命令来列举当前存在的 Topic。

```
[root@ localhost bin]# pwd
/usr/dahua/kafka/bin
[root@ localhost bin]# ./kafka-topics.sh --zookeeper
  localhost:2181 --list
test
```

图 6-6 列举 Topic

其中：

- --list 设置此次操作的 action 类型为列举。
- --zookeeper localhost:2181 设置 Zookeeper 集群的地址。

listTopics 就是列举当前 Kafka 集群中存在的 Topic，本质上就是从路径为 /brokers/topics/ 的 Zookeeper 目录下加载当前集群中存在的 Topic，然后从路径为 /admin/delete_topics/ 的 Zookeeper 目录下标记当前集群中正在删除的 Topic，即具体实现如下：

```
def listTopics(zkClient: ZkClient, opts: TopicCommandOptions) {
  // 从 /brokers/topics/ 加载当前存在的 Topic 列表
  val topics = getTopics(zkClient, opts)
  for(topic <- topics) {
    // 从 /admin/delete_topics/ 目录标记当前正在删除的 Topic
    if (ZkUtils.pathExists(zkClient,ZkUtils.getDeleteTopicPath(topic))) {
      // 输出正在删除的 Topic
      println("%s - marked for deletion".format(topic))
    } else {
      // 输出正常的 Topic
      println(topic)
    }
  }
}
```

6.1.4　describeTopic

在 Kafka 安装包的 bin 目录下执行图 6-7 所示的命令来查看指定 Topic 的信息。

```
[root@ localhost bin]# pwd
/usr/dahua/kafka/bin
[root@ localhost bin]# ./kafka-topics.sh --zookeeper localhost:2181
--describe --topic test
Topic:test      PartitionCount:4      ReplicationFactor:2
  Configs:flush.messages=1
      Topic: test    Partition: 0   Leader: 3      Replicas: 3,1  Isr: 3,1
      Topic: test    Partition: 1   Leader: 4      Replicas: 4,2  Isr: 4,2
      Topic: test    Partition: 2   Leader: 3      Replicas: 3,1  Isr: 3,1
      Topic: test    Partition: 3   Leader: 4      Replicas: 4,2  Isr: 4,2
```

图 6-7　描述 Topic

其中：

- --describe 设置此次操作的 action 类型为描述。
- --zookeeper localhost:2181 设置 Zookeeper 集群的地址。
- --topic 设置 topic 的名称。

describeTopic 就是把 Topic 的详细信息展示出来。Topic 的详细信息包括：

- Topic 的 Replica 分布情况（Partition 的 AR 列表），其位于路径为 /brokers/topics/[topic] 的 Zookeeper 目录下。
- Topic 的 Config 信息，其位于路径为 /config/topics/[topic] 的 Zookeeper 目录下。
- Topic 的分区状态，其位于 /brokers/topics/[topic]/partitions/[partitionId]/[state] 的 Zookeeper 目录下。

其具体代码实现如下：

```
def describeTopic(zkClient: ZkClient, opts: TopicCommandOptions) {
  /* 如果指定了 --topic 参数，则获取指定的 Topic，如果没有指定，则从 /brokers/topics/ 获取全部
     的 Topic*/
  val topics = getTopics(zkClient, opts)
  // 是否输出正在处于复制状态的 Partition，即 Partition 的 ISR 列表和 Partition 的 AR 列表不相等
  val reportUnderReplicatedPartitions = if (opts.options.has(opts.reportUnderR
    eplicatedPartitionsOpt)) true
    else
      false
  // 是否输出不可用的 Partition 的 Topic
  val reportUnavailablePartitions = if (opts.options.has(opts.reportUnavailabl
    ePartitionsOpt))
      true
    else
      false
  // 是否输出 Topic 的 Config 被重写过的 Topic
```

```
val reportOverriddenConfigs = if (opts.options.has(opts.topicsWithOverridesOpt))
    true
  else
    false
//获取在线的 Broker 列表
val liveBrokers = ZkUtils.getAllBrokersInCluster(zkClient).map(_.id).toSet
for (topic <- topics) {
  //从 /brokers/topics/[topic] 目录加载 Partition 的 AR 列表
  ZkUtils.getPartitionAssignmentForTopics(zkClient, List(topic)).get(topic) match {
    case Some(topicPartitionAssignment) =>
      //判断是否描述 Topic 的 Config
      val describeConfigs: Boolean = !reportUnavailablePartitions &&
        !reportUnderReplicatedPartitions
      //判断是否描述 Topic 的 Partition
      val describePartitions: Boolean = !reportOverriddenConfigs
      //将 Partition 排序
      val sortedPartitions = topicPartitionAssignment.toList.sortWith((m1,
        m2) => m1._1 < m2._1)
      if (describeConfigs) {
        //从 /config/topics/[topic] 目录加载 Topic 的 Config
        val configs = AdminUtils.fetchTopicConfig(zkClient, topic)
        /* 如果 configs.size==0，则说明 Topic 的 Config 被 overridden，则根据
           reportOverriddenConfigs 判断是否输出 */
        if (!reportOverriddenConfigs || configs.size() != 0) {
          val numPartitions = topicPartitionAssignment.size
          val replicationFactor = topicPartitionAssignment.head._2.size
          //输出分区个数和副本因子
          println("Topic:%s\tPartitionCount:%d\tReplicationFactor:%d\tConfigs:%s"
            .format(topic, numPartitions, replicationFactor, configs.map(kv
              => kv._1 + "=" + kv._2).mkString(",")))
        }
      }
      if (describePartitions) {
        for ((partitionId, assignedReplicas) <- sortedPartitions) {
          //从 /brokers/topics/[topic]/partitions/[partitionId]/[state] 目录加
            载分区状态信息
          val inSyncReplicas = ZkUtils.getInSyncReplicasForPartition(zkClie
            nt, topic, partitionId)
          val leader = ZkUtils.getLeaderForPartition(zkClient, topic, partitionId)
          /*
            满足以下条件中的某一个就输出：
            1）仅仅是正常输出，即 reportUnderReplicatedPartitions 和
               reportUnavailablePartitions 为 false
            2）输出正在处于复制状态的 Partition，则此时 ISR 列表的个数肯定小于 AR 列表的个数
            3）输出分区不可用的 Topic，则此时 ISR 中的 Leader 不存在或者下线
          */
          if ((!reportUnderReplicatedPartitions &&
            !reportUnavailablePartitions) ||
              (reportUnderReplicatedPartitions && inSyncReplicas.size <
                assignedReplicas.size) ||
              (reportUnavailablePartitions &&
```

```
                (!leader.isDefined || !liveBrokers.contains(leader.get))) {
              // 输出 Topic, PartitionId, Leader, Replicas, ISR
              print("\tTopic: " + topic)
              print("\tPartition: " + partitionId)
              print("\tLeader: " + (if(leader.isDefined) leader.get else "none"))
              print("\tReplicas: " + assignedReplicas.mkString(","))
              println("\tIsr: " + inSyncReplicas.mkString(","))
            }
          }
        }
      case None =>
        // 否则输出 Topic 不存在
        println("Topic " + topic + " doesn't exist!")
    }
  }
}
```

6.1.5　deleteTopic

在 Kafka 安装包的 bin 目录下执行图 6-8 所示的命令来删除 Topic。

```
[root@ localhost bin]# pwd
/usr/dahua/kafka/bin
[root@ localhost bin]# ./kafka-topics.sh --zookeeper
  localhost:2181 --delete --topic test
Topic test is marked for deletion.
Note: This will have no impact if delete.topic.enable is not set to true.
```

图 6-8　删除 Topic

其中：

- --delete 设置此次操作的 action 类型为删除。
- --zookeeper localhost:2181 设置 Zookeeper 集群的地址。
- --topic 设置 topic 的名称。

deleteTopic 本质上就是将需要待删除的 Topic 持久化至路径为 /admin/delete_topics/[topic] 的 Zookeeper 目录上，其具体实现如下：

```
def deleteTopic(zkClient: ZkClient, opts: TopicCommandOptions) {
  /* 如果指定了 --topic 参数，则获取指定的 Topic，如果没有指定，则从 /brokers/topics/ 获取全部
     的 Topic*/
  val topics = getTopics(zkClient, opts)
  // 判断 Topic 是否存在
  if (topics.length == 0) {
    println("Topic %s does not exist".format(opts.options.valueOf(opts.
      topicOpt)))
  }
  // 遍历 Topic 列表
  topics.foreach { topic =>
```

```
        try {
          // 将 Topic 持久化至 /admin/delete_topics/[topic]，标记删除
          ZkUtils.createPersistentPath(zkClient, ZkUtils.getDeleteTopicPath(topic))
          println("Topic %s is marked for deletion.".format(topic))
          println("Note: This will have no impact if delete.topic.enable is not
            set to true.")
        } catch {
          // 由于删除 Topic 是异步流程，则如果之前节点存在，则说明还正在删除中
          case e: ZkNodeExistsException =>
            println("Topic %s is already marked for deletion.".format(topic))
          // 其他异常
          case e2: Throwable =>
            throw new AdminOperationException("Error while deleting topic %s".format(topic))
        }
      }
    }
```

当 deleteTopic 完成以上步骤之后，就会触发 5.6.7 小节中提到的 DeleteTopicsListener 监听器的删除 Topic 流程，最终完成 Topic 的删除。

6.2 kafka-reassign-partitions.sh

kafka-reassign-partitions.sh 提供了重新分配分区副本的能力。该工具可以促进 Kafka 集群的负载均衡。因为 Follower Replica 需要从 Leader Replica Fetch 数据以保持与 Leader Replica 同步，所以仅仅保持 Leader Replica 分布的平衡对整个集群的负载均衡来说是不够的。另外，生产环境下，随着负载的增大，可能需要给 Kafka 集群扩容。向 Kafka 集群中增加 Broker 非常简单方便，但是对于已有的 Topic，并不会自动将其 Partition 迁移到新加入的 Broker 上，此时可用该工具达到此目的。除此之外在某些场景下，实际负载可能远小于最初预期负载，此时可用该工具将分布在整个集群上的 Partition 重装分配到某些机器上，然后可以停止不需要的 Broker 从而实现节约资源的目的。

在内部是通过 ReassignPartitionsCommand 类来实现功能的，其脚本的内容如下：

```
/*kafka-run-class.sh 加载 kafka 的 classpath，执行其中的 kafka.admin.
  ReassignPartitionsCommand 类的 main 函数 */
exec $(dirname $0)/kafka-run-class.sh kafka.admin.ReassignPartitionsCommand $@
```

通过解析不同的 action 参数来执行不同的流程，如下所示：

```
object ReassignPartitionsCommand extends Logging {
  def main(args: Array[String]): Unit = {
    // 解析参数
    val opts = new ReassignPartitionsCommandOptions(args)
    // 判断是否有多个 action 参数
    val actions = Seq(opts.generateOpt, opts.executeOpt, opts.verifyOpt).
      count(opts.options.has _)
```

```
    // 一次只支持一种 action，否则命令无效
    if(actions != 1)
      CommandLineUtils.printUsageAndDie(
        opts.parser,
        "Command must include exactly one action: --generate, --execute or --verify")
    // 检查 --zookeeper 参数是否存在
    CommandLineUtils.checkRequiredArgs(
      opts.parser,
      opts.options,
      opts.zkConnectOpt)
    // 提取 --zookeeper 参数
    val zkConnect = opts.options.valueOf(opts.zkConnectOpt)
    // 创建 Zookeeper 连接
    var zkClient: ZkClient = new ZkClient(
      zkConnect,
      30000,
      30000,
      ZKStringSerializer)
    try {
      if(opts.options.has(opts.verifyOpt))
        // 包含 verify 关键字，则验证重新分配 Partition 是否成功
        verifyAssignment(zkClient, opts)
      else if(opts.options.has(opts.generateOpt))
        /* 包含 generate 关键字，则给定需要重新分配的 Topic，自动生成 Reassign Plan，但是
           并不执行 */
        generateAssignment(zkClient, opts)
      else if (opts.options.has(opts.executeOpt))
        // 包含 execute 关键字，则根据指定的 Reassign Plan 重新分配 Partition
        executeAssignment(zkClient, opts)
    } catch {
      case e: Throwable =>
        println("Partitions reassignment failed due to " + e.getMessage)
        println(Utils.stackTrace(e))
    } finally {
      // 关闭 Zookeeper 连接
      if (zkClient != null)
        zkClient.close()
    }
  }
}
```

其中支持以下若干个 action 参数：

- --generate 生成分区重分配计划。
- -- excute 执行分区重分配计划。
- --verify 验证分区重分配计划。

通过以上三种不同的 action 类型，可以实现 Topic 分区重分配的管理。接下来将详细描述以上 action 的控制台输出和内部详细的执行原理。

6.2.1 generateAssignment

在 Kafka 安装包的 bin 目录下执行图 6-9 所示的命令来生成 Reassign Plan。

```
[root@ localhost bin]# pwd
/usr/dahua/kafka/bin
[root@ localhost bin]# ./kafka-reassign-partitions.sh --zookeeper
  localhost:2181 --topics-to-move-json-file /tmp/topics-to-move.json
  --broker-list "2,3" --generate
Current partition replica assignment
{"version":1,"partitions":[{"topic":"test","partition":0,"replicas":[4,1]}]}
Proposed partition reassignment configuration
{"version":1,"partitions":[{"topic":"test","partition":0,"replicas":[2,3]}]}
```

图 6-9 生成 Reassign Plan

其中：

- --generate 设置此次操作的 action 类型为产生 Reassign Plan。
- --zookeeper localhost:2181 设置 Zookeeper 集群的地址。
- --topics-to-move-json-file 指定 JSON 格式的配置文件，格式为：（{"topics":[{"topic":"test"}],"version":1}），topic 指定重分配的 Topic 名称。
- --broker-list 设置 Partition 重分配的 Broker Server 范围。

其具体实现代码如下：

```
def generateAssignment(zkClient: ZkClient,
                       opts: ReassignPartitionsCommandOptions) {
  // 判断 --topics-to-move-json-file 参数和 --broker-list 是否存在
  if(!(opts.options.has(opts.topicsToMoveJsonFileOpt)
    && opts.options.has(opts.brokerListOpt)))
    CommandLineUtils.printUsageAndDie(
      opts.parser,
      "If --generate option is used, command must include both --topics-to-
        move-json-file and --broker-list options")
  // 获取配置文件地址
  val topicsToMoveJsonFile = opts.options.valueOf(opts.topicsToMoveJsonFileOpt)
  // 获取 Broker 列表
  val brokerListToReassign = opts.options.valueOf(opts.brokerListOpt).
    split(',').map(_.toInt)
  // 提取重复的 Broker
  val duplicateReassignments = Utils.duplicates(brokerListToReassign)
  // 判断 Broker 列表是否包含重复的 Broker
  if (duplicateReassignments.nonEmpty)
    throw new AdminCommandFailedException("Broker list contains duplicate
      entries: %s".format(duplicateReassignments.mkString(",")))
  // 读取配置文件
```

```
    val topicsToMoveJsonString = Utils.readFileAsString(topicsToMoveJsonFile)
    // 解析出需要充分配的 Topic
    val topicsToReassign = ZkUtils.parseTopicsData(topicsToMoveJsonString)
    // 提取重复的 Topic
    val duplicateTopicsToReassign = Utils.duplicates(topicsToReassign)
    // Topic 列表是否包含重复的 Topic
    if (duplicateTopicsToReassign.nonEmpty)
      throw new AdminCommandFailedException("List of topics to reassign contains
        duplicate entries: %s".format(duplicateTopicsToReassign.mkString(",")))
    // 获取待重新分配的 Partition
    val topicPartitionsToReassign = ZkUtils.getReplicaAssignmentForTopics(zkClie
      nt, topicsToReassign)
    var partitionsToBeReassigned : Map[TopicAndPartition, Seq[Int]] =
      new mutable.HashMap[TopicAndPartition, List[Int]]()
    // 将 Partition 按照 Topic 分组
    val groupedByTopic = topicPartitionsToReassign.groupBy(tp => tp._1.topic)
    groupedByTopic.foreach { topicInfo =>
      // 将相同 Topic 下的 Partition 利用 Kafka 集群提供的默认负载均衡算法进行重分配
      val assignedReplicas = AdminUtils.assignReplicasToBrokers(
        brokerListToReassign,
        topicInfo._2.size,
        topicInfo._2.head._2.size)
      // 保存重新分配的结果
      partitionsToBeReassigned ++= assignedReplicas.map(
        replicaInfo => (TopicAndPartition(topicInfo._1, replicaInfo._1) ->
          replicaInfo._2))
    }
    // 获取当前待分配的 Partition 的 AR 列表
    val currentPartitionReplicaAssignment = ZkUtils.getReplicaAssignmentForTopics
      (zkClient,
      partitionsToBeReassigned.map(_._1.topic).toSeq)
    // 打印当前的 Partition Replica 分布情况
    println("Current partition replica assignment\n\n%s"
      .format(ZkUtils.getPartitionReassignmentZkData(currentPartitionReplicaAssi
        gnment)))
    // 打印提议的 Partition Replica 分布情况
    println("Proposed partition reassignment configuration\n\n%s".
      format(ZkUtils.getPartitionReassignmentZkData(partitionsToBeReassigned)))
  }
```

当 ./kafka-reassign-partitions.sh --zookeeper localhost:2181 --topics-to-move-json-file /tmp/topics-to-move.json --broker-list "2,3" -generate 执行完毕之后，保存其输出的关于 Proposed partition reassignment 打印，将其保存为 reassign-plan.json。

6.2.2　executeAssignment

在 Kafka 安装包的 bin 目录下执行图 6-10 所示的命令来执行 Reassign Plan。

```
[root@ localhost bin]# pwd
/usr/dahua/kafka/bin
[root@ localhost bin]# ./kafka-reassign-partitions.sh --zookeeper
  localhost:2181
--reassignment-json-file /tmp/reassign-plan.json --execut
Current partition replica assignment
{"version":1,"partitions":[{"topic":"test","partition":0,"replicas":[4,1]}]}
Save this to use as the --reassignment-json-file option during rollback
Successfully started reassignment of partitions
{"version":1,"partitions":[{"topic":"test","partition":0,"replicas":[2,3]}]}
```

图 6-10 执行 Reassign Plan

其中：

- --execut 设置此次操作的 action 类型为执行 Reassign Plan。
- --zookeeper localhost:2181 设置 Zookeeper 集群的地址。
- --reassignment-json-file 指定 JSON 格式的配置文件，里面阐述了 Topic 分区重分配之后的 AR 情况。

其具体实现如下：

```
def executeAssignment(zkClient: ZkClient,
                      opts: ReassignPartitionsCommandOptions) {
  //reassignment-json-file 参数是否存在
  if(!opts.options.has(opts.reassignmentJsonFileOpt))
    CommandLineUtils.printUsageAndDie(
      opts.parser,
      "If --execute option is used, command must include --reassignment-json-
        file that was output " +
        "during the --generate option")
  //获取 /reassignment-json-file 参数指向的文件地址
  val reassignmentJsonFile = opts.options.valueOf(opts.reassignmentJsonFileOpt)
  //读取该配置文件
  val reassignmentJsonString = Utils.readFileAsString(reassignmentJsonFile)
  //解析该分区重分配文件
  val partitionsToBeReassigned = ZkUtils.parsePartitionReassignmentDataWithout
    Dedup(reassignmentJsonString)
  //判断待重分配的 Partition 个数是否为 0
  if(partitionsToBeReassigned.isEmpty)
    throw new AdminCommandFailedException(
      "Partition reassignment data file %s is empty".format(reassignmentJsonFile))
  //提取重复重分配的 Partition
  val duplicateReassignedPartitions = Utils.duplicates(partitionsToBeReassign
    ed.map{case(tp,replicas) => tp})
  //判断是否存在相同的 Partition 多次重分配
  if (duplicateReassignedPartitions.nonEmpty)
    throw new AdminCommandFailedException("Partition reassignment contains
      duplicate topic
```

```
      partitions: %s".format(duplicateReassignedPartitions.mkString(",")))
    // 提取 Partition 的 AR 列表，其 AR 列表可能存在重复的 Replica
    val duplicateEntries= partitionsToBeReassigned
      .map{ case(tp,replicas) => (tp, Utils.duplicates(replicas))}
      .filter{ case (tp,duplicatedReplicas) => duplicatedReplicas.nonEmpty }
    // 判断 Partition 的 AR 列表是否存在重复的 Replica
    if (duplicateEntries.nonEmpty) {
      val duplicatesMsg = duplicateEntries.map{
        case (tp,duplicateReplicas) => "%s contains multiple entries for %s".format(
          tp,
          duplicateReplicas.mkString(",")) }
        .mkString(". ")
      throw new AdminCommandFailedException("Partition replica lists may not
        contain duplicate entries: %s".format(duplicatesMsg))
    }
    val reassignPartitionsCommand = new ReassignPartitionsCommand(
      zkClient,
      partitionsToBeReassigned.toMap)
    // 获取当前的 Partition 的 AR 列表
    val currentPartitionReplicaAssignment = ZkUtils.getReplicaAssignmentForTopics
      (zkClient, partitionsToBeReassigned.map(_._1.topic).toSeq)
    // 打印当前的 Partition 的 AR 列表，如果重分配失败，则可以利用此信息进行回滚
    println("Current partition replica assignment\n\n%s\n\nSave this to use as
      the --reassignment-json-file option during rollback"
      .format(ZkUtils.getPartitionReassignmentZkData
        (currentPartitionReplicaAssignment)))
    // 执行重分配的动作
    if(reassignPartitionsCommand.reassignPartitions())
      println("Successfully started reassignment of partitions %s".format(ZkUtils.
        getPartitionReassignmentZkData(partitionsToBeReassigned.toMap)))
    else
      println("Failed to reassign partitions %s".format(partitionsToBeReassigned))
  }
```

回想 5.6.3 小节中的 PartitionsReassignedListener 监听器，监听到路径为 /admin/reassign_partitions 的 Zookeeper 目录上发生数据变化时，会加载该路径下设置的 Topic 分区重分配信息，执行真正的分区重分配，因此 ReassignPartitionsCommand 的 reassignPartitions 方法正是将 Topic 分区的重分配信息写入 /admin/reassign_partitions 目录，其具体实现如下：

```
class ReassignPartitionsCommand(zkClient: ZkClient,
                                partitions: collection.Map[TopicAndPartition,
                                collection.Seq[Int]])
  extends Logging {
  def reassignPartitions(): Boolean = {
    try {
      // 过滤出有效的 Partition
      val validPartitions = partitions.filter(p => validatePartition(zkClient,
        p._1.topic, p._1.partition))
      // 将其转化为字符串，以便写入 Zookeeper
```

```
        val jsonReassignmentData = ZkUtils.getPartitionReassignmentZkData
          (validPartitions)
        //在 /admin/reassign_partitions 目录持久化此重分配信息
        ZkUtils.createPersistentPath(zkClient, ZkUtils.ReassignPartitionsPath,
          jsonReassignmentData)
        true
      } catch {
        //当前集群正在进行其他分区的重分配，不支持同时执行
        case ze: ZkNodeExistsException =>
          val partitionsBeingReassigned = ZkUtils.getPartitionsBeingReassigned
            (zkClient)
          throw new AdminCommandFailedException("Partition reassignment
          currently in " +
            "progress for %s. Aborting operation".format(partitionsBeingReassigned))
        //其他异常
        case e: Throwable => error("Admin command failed", e); false
      }
    }
  }
```

分区重分配的具体执行细节请参考 5.6.3PartitionsReassignedListener 小节中的内容。

6.2.3 verifyAssignment

由于分区重分配是一个异步的流程，因此 kafka-reassign-partitions.sh 脚本提供了查看当前分区重分配进度的指令。

在 Kafka 安装包的 bin 目录下执行图 6-11 所示的命令来查看 Reassign Plan 执行情况。

```
[root@ localhost bin]# pwd
/usr/dahua/kafka/bin
[root@ localhost bin]# ./kafka-reassign-partitions.
  sh --zookeeperlocalhost:2181
--reassignment-json-file /tmp/reassign-plan.json --verify
Status of partition reassignment:
Reassignment of partition [test,0] completed successfully
```

图 6-11 查看 Reassign Plan 执行情况

其中：

- --execut 设置此次操作的 action 类型为查看 Reassign Plan 进度。
- --zookeeper localhost:2181 设置 Zookeeper 集群的地址。
- --reassignment-json-file 指定 JSON 格式的配置文件，里面阐述了 Topic 分区重分配之后的 AR 情况。

其具体实现代码如下：

```
def verifyAssignment(zkClient: ZkClient,
                     opts: ReassignPartitionsCommandOptions) {
```

```
    //判断是否包含 --reassignment-json-file 参数
    if(!opts.options.has(opts.reassignmentJsonFileOpt))
      CommandLineUtils.printUsageAndDie(
        opts.parser,
        "If --verify option is used, command must include --reassignment-json-
          file that was used during the --execute option")
    //获取指定的配置文件地址
    val jsonFile = opts.options.valueOf(opts.reassignmentJsonFileOpt)
    //读取配置文件
    val jsonString = Utils.readFileAsString(jsonFile)
    //解析配置文件
    val partitionsToBeReassigned = ZkUtils.parsePartitionReassignmentData(jsonString)
    println("Status of partition reassignment:")
    //获取每个 Partition 的重分配状态
    val reassignedPartitionsStatus = checkIfReassignmentSucceeded(zkClient,
      partitionsToBeReassigned)
    //打印状态，方便用户在控制台可以看到
    reassignedPartitionsStatus.foreach { partition =>
      partition._2 match {
        case ReassignmentCompleted =>
          println("Reassignment of partition %s completed successfully".format(partition._1))
        case ReassignmentFailed =>
          println("Reassignment of partition %s failed".format(partition._1))
        case ReassignmentInProgress =>
          println("Reassignment of partition %s is still in progress".format(partition._1))
      }
    }
  }
```

那么究竟是如何来实现查看分区重分配状况的呢？也就是说分区的重分配进度是如何体现出来的呢？回想5.6.3小节中的PartitionsReassignedListener监听器，会监听路径为/admin/reassign_partitions的Zookeeper目录，在这个目录保存了需要待重分配的分区详情集合。当某个分区完成重新分配之后，则会把对应的重分配分区详情从/admin/reassign_partitions目录下删除。因此只要分区包含在/admin/reassign_partitions目录下，则说明分区重分配正在进行中，如果不包含在/admin/reassign_partitions目录下，则说明分区重分配已经完成，其具体执行情况需要比对分区当前的AR列表和待重分配的AR列表，如果两者相同，则说明成功；否则说明失败。checkIfReassignmentSucceeded的具体实现如下：

```
  def checkIfReassignmentSucceeded(
    zkClient: ZkClient,
    partitionsToBeReassigned: Map[TopicAndPartition, Seq[Int]])
    :Map[TopicAndPartition, ReassignmentStatus] = {
    //从 /admin/reassign_partitions 目录获取正在进行重分配的分区列表
    val partitionsBeingReassigned = ZkUtils.getPartitionsBeingReassigned(zkClie  nt).
      mapValues(_.newReplicas)
    //根据待分配的分区列表和正在进行分配的分区列表判断分区重分配的执行情况
    partitionsToBeReassigned.map { topicAndPartition =>
```

```
      (topicAndPartition._1,
      checkIfPartitionReassignmentSucceeded(
        zkClient,
        topicAndPartition._1,
        topicAndPartition._2,
        partitionsToBeReassigned,
        partitionsBeingReassigned))
    }
  }
  def checkIfPartitionReassignmentSucceeded(
    zkClient: ZkClient,
    topicAndPartition: TopicAndPartition,
    reassignedReplicas: Seq[Int],
    partitionsToBeReassigned: Map[TopicAndPartition, Seq[Int]],
    partitionsBeingReassigned: Map[TopicAndPartition, Seq[Int]]):
      ReassignmentStatus = {
    // 获取待分配分区的 AR 列表
    val newReplicas = partitionsToBeReassigned(topicAndPartition)
    partitionsBeingReassigned.get(topicAndPartition) match {
      // 正在进行重分配的分区中包含该分区，则表明分区正处于重分配状态
      case Some(partition) => ReassignmentInProgress
      // 正在进行重分配的分区中没有包含该分区，则表明分区重分配已经结束
      case None =>
        // 获取当前的 AR 列表
        val assignedReplicas = ZkUtils.getReplicasForPartition(
          zkClient,
          topicAndPartition.topic,
          topicAndPartition.partition)
        // 如果当前 AR 列表和待重分配的 AR 列表相同，则说明重分配成功，否则说明失败
        if(assignedReplicas == newReplicas)
          ReassignmentCompleted
        else {
          println(("ERROR: Assigned replicas (%s) don't match the list of replicas
            for reassignment (%s)" +" for partition %s").format(assignedReplicas.
            mkString(","), newReplicas.mkString(","), topicAndPartition))
          ReassignmentFailed
        }
    }
  }
```

6.3 kafka-preferred-replica-election.sh

Topic 由若干个 Partition 组成，每个 Partition 又由若干个 Replica 组成。每个 Partition 的第一个 Replica 称为“Preferred Replica”，回顾 6.1.1 节提到的 Topic 的 Replica 分配算法，会尽可能将 Topic 的所有 Replica 均匀地分布在 Kafka 集群中，并且每个 Partition 的 Leader Replica 为其 Preferred Replica，这样就保证了 Leader Replica 带来的负载在整个集群中是均衡的。然后，如果有 Broker Shutdown 了（比如 crash，机器故障）的话，那么

Leader Replica 带来的负载就不均衡了。此工具就是用来在整个集群中恢复 Leader Replica 为 Preferred Replica。

kafka-preferred-replica-election.sh 脚本提供了触发 Topic 的 Partition 进行 Preferred Replica Election 的能力，内部是通过 PreferredReplicaLeaderElectionCommand 类来实现功能的，其脚本的内容如下：

```
// kafka-run-class.sh 加载 kafka 的 classpath，执行其中的 kafka.admin.
// PreferredReplicaLeaderElectionCommand 类的 mian 函数
exec $(dirname $0)/kafka-run-class.sh kafka.admin.
  PreferredReplicaLeaderElectionCommand $@
```

在 Kafka 安装包的 bin 目录下执行图 6-12 所示的命令来执行 Preferred Replica Election。

```
[root@ localhost bin]# pwd
/usr/dahua/kafka/bin
[root@ localhost bin]# ./kafka-preferred-replica-election.
  sh --zookeeper localhost:2181 --path-to-json /tmp/
  topicPartitionList.json
Successfully started preferred replica election for partitions
  Set([test,0])
```

图 6-12　执行 Preferred Replica Election

其中：

- --zookeeper localhost:2181 设置 Zookeeper 集群的地址。
- --path-to-json-file 指定 JSON 格式的配置文件，里面包含了 Topic 的 Partition 列表，如果不输入该参数的话，则会对当前所有的 Topic 执行。

图 6-13 是 --path-to-json-file 指定的 JSON 格式文件的例子。

```
{
"partitions":
  [
    {"topic": "test", "partition": "0"},
    {"topic": "test", "partition": "1"},
    {"topic": "test", "partition": "2"},
    {"topic": "test", "partition": "3"}
  ]
}
```

图 6-13　JSON 示例文件

当执行完此命令之后，就会在路径为 /admin/preferred_replica_election 的 Zookeeper 目录上持久化需要执行的 Partition 列表，其具体实现如下：

```
object PreferredReplicaLeaderElectionCommand extends Logging {
  def main(args: Array[String]): Unit = {
```

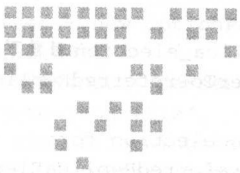
```
//创建解析器
val parser = new OptionParser
//设置需要解析的项
val jsonFileOpt = parser.accepts(
  "path-to-json-file",
  "The JSON file with the list of partitions " +
  "for which preferred replica leader election should be done, in the following
    format - \n" +
    "{\"partitions\":\n\t[{\"topic\": \"foo\", \"partition\": 1},\n\t
      {\"topic\": \"foobar\", \"partition\": 2}]\n}\n" +"Defaults to all existing
        partitions")
  .withRequiredArg
  .describedAs("list of partitions for which preferred replica leader election
    needs to be triggered")
  .ofType(classOf[String])
//设置需要解析的项
val zkConnectOpt = parser.accepts(
  "zookeeper",
  "REQUIRED: The connection string for the zookeeper connection in the " +
    "form host:port. Multiple URLS can be given to allow fail-over.")
  .withRequiredArg
  .describedAs("urls")
  .ofType(classOf[String])
//如果参数为 0，则打印使用方法并退出
if(args.length == 0)
  CommandLineUtils.printUsageAndDie(
    parser,
    "This tool causes leadership for each partition to be transferred back to
      the 'preferred
      replica'," + " it can be used to balance leadership among the servers.")
//解析指定参数
val options = parser.parse(args : _*)
//判断是否包含 zookeeper 选项
CommandLineUtils.checkRequiredArgs(parser, options, zkConnectOpt)
//获取 zookeeper 选项内容
val zkConnect = options.valueOf(zkConnectOpt)
var zkClient: ZkClient = null
try {
  //根据 zookeeper 选项内容创建 Zookeeper 客户端
  zkClient = new ZkClient(zkConnect, 30000, 30000, ZKStringSerializer)
  val partitionsForPreferredReplicaElection =
    if (!options.has(jsonFileOpt))
      //不包含 path-to-json-file 选项，则从 /brokers/topics/ 目录下加载所有的 Partition
      ZkUtils.getAllPartitions(zkClient)
    Else
      //包含 path-to-json-file 选项，则从 path-to-json-file 指向的配置文件中加载 Partition
      parsePreferredReplicaElectionData(Utils.readFileAsString(options.
        valueOf(jsonFileOpt)))
  val preferredReplicaElectionCommand = new PreferredReplicaLeaderElection
    Command(
```

```
            zkClient,
            partitionsForPreferredReplicaElection)
          // 将 Partition 持久化至 /admin/preferred_replica_election 目录下
          preferredReplicaElectionCommand.moveLeaderToPreferredReplica()
          println(
            "Successfully started preferred replica election for
              partitions %s".format(partitionsForPreferredReplicaElection))
        } catch {
          case e: Throwable =>
            println("Failed to start preferred replica election")
            println(Utils.stackTrace(e))
        } finally {
          // 关闭 Zookeeper 连接
          if (zkClient != null)
            zkClient.close()
        }
      }
    }
```

一旦 PreferredReplicaLeaderElectionCommand 将需要进行 Preferred Replica Election 的 Partition 持久化至 /admin/preferred_replica_election 的 Zookeeper 目录上之后，会触发 5.6.5 小节提到的 PreferredReplicaElectionListener 监听器，从而触发 Partition 的 Preferred Replica Election。

6.4　本章小结

本章主要讲解了有关 Topic 管理工具的使用方法和内部实现原理，主要有三个脚本：kafka-topics.sh、kafka-reassign-partitions.sh 和 kafka-preferred-replica-election.sh。这三个脚本提供了涉及 Topic 生命周期管理的所有功能，作为一个合格的 Kafka 的使用者，不仅需要知道如何使用这三个脚本去维护 Topic，更重要的是需要深刻理解其内部的实现原理，也就是需要知道维护脚本、Zookeeper 和 KafkaController 内部的监听器这三者之间的关系。



Chapter 7 第 7 章

生 产 者

生产者是指消息的生成者，即将消息发送到指定的 Topic 中的生成者。生产者可以通过特定的分区函数决定消息路由到 Topic 的某个分区。消息的生成者发送消息有两种模式，分别为同步模式和异步模式。

本章先从生产者的设计原则讲起，然后结合生产者的客户端代码描述如何发送消息，最后深入内部的具体细节，讲述生产者的模块组成以及不同模式下消息发送的特点。

7.1 设计原则

生产者就是将消息发送到指定的 Topic 中。回顾 4.3.5.1 ProducerRequest 小节，生产者本质上就是指定具体的 Topic，然后向目的端 Broker Server 发送 ProducerRequest 请求，并且通过分区函数可以路由具体的消息至特定的分区。生产者内部会动态维护与 Topic 相关的 Broker Server 的 Socket 链接，客户端无需手动维护。因此生产者具备以下特点：客户端消息发送时只需要指定 Topic 和消息即可，不需要指定目的端 Broker Server。

生产者发送消息的简要流程如图 7-1 所示。

7.2 示例代码

参考图 7-1 生产者消息发送的简要流程，客户端编程时只需要获取 Kafka 提供的生产者对象，然后指定具体的 Topic 和具体的消息即可将消息发送到 Topic 的某个分区，如果需要额外控制消息的路由规则，则需要额外提供分区函数和分区键，使得分区函数通过分区键将

消息进行路由。

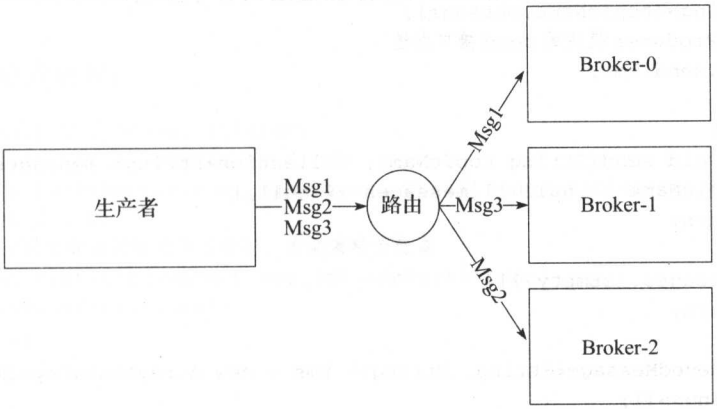


图 7-1 生产者消息发送的简要流程

Kafka 提供的生产者对象为 kafka.producer，Kafka 提供的消息对象为 kafka.producer.KeyedMessage，通过 kafka.producer 提供的 send 接口可以单条发送消息和批量发送消息。因此客户端编程时只需要获取 kafka.producer 对象，然后组装 kafka.producer.KeyedMessage 消息，最后利用 kafka.producer 提供的 send 接口将 kafka.producer.KeyedMessage 发送出去，其生产者的具体示例代码如下所示：

```
import kafka.javaapi.producer.Producer;
import kafka.producer.KeyedMessage;
import kafka.producer.ProducerConfig;
import java.util.*;
public class KafkaProducer {
  //生产者，对应的类为 kafka.javaapi.producer.Producer
  private Producer<String,String> inner;
  public KafkaProducer() throws Exception{
    Properties props = new Properties();
    //配置 Broker 地址
    props.put("metadata.broker.list", "172.23.9.133:9092,172.23.9.134:9092,
      172.23.9.135:9092");
    //配置分区函数，分区函数决定消息的路由
    props.put("partitioner.class", "com.dahua.kafka.KafkaPartitioner");
    //通过 Properties 生成 ProducerConfig
    ProducerConfig config = new ProducerConfig(props);
    //初始化 Producer
    inner = new Producer<String, String>(config);
  }
  //单条发送
  public void send(String topicName, String message) {
    if(topicName == null || message == null){
      return;
    }
```

```
        //消息由 KeyedMessage 表示
        KeyedMessage<String, String> km = new KeyedMessage<String,
          String>(topicName,message);
        //通过 Producer 提供的 send 接口发送
        inner.send(km);
      }
      //批量发送
      public void send(String topicName, Collection<String> messages) {
        if(topicName == null || messages == null){
          return;
        }
        if(messages.isEmpty()){
          return;
        }
        List<KeyedMessage<String, String>> kms = new ArrayList<KeyedMessage<String,
          String>>();
        for(String entry : messages){
          //消息由 KeyedMessage 表示
          KeyedMessage<String, String> km = new KeyedMessage<String,
            String>(topicName,entry);
          kms.add(km);
        }
        //通过 Producer 提供的 send 接口发送
        inner.send(kms);
      }
      public void close(){
        //关闭生成者
        inner.close();
      }
      public static void main(String[] args) {
        KafkaProducer producer = null;
        try{
          producer = new KafkaProducer();
          int i=0;
          while(true){
            StringBuffer sbMsg = new StringBuffer();
            sbMsg.append("content");
            //以 KV 对的形式发送
            producer.send("key", sbMsg.toString());
          }
        }catch(Exception e){
          e.printStackTrace();
        }finally{
          //无论如何都需要释放资源，关闭 producer
          if(producer != null){
            producer.close();
          }
        }
      }
    }
```

Producer 在初始化过程中通过 metadata.broker.list 指定了 Kafka 集群内的 Broker Server 地址，通过 partitioner.class 指定了分区函数的地址，涉及的其他参数在接下来小节中将分别一一阐述。

分区函数的形式如下：

```
import kafka.producer.Partitioner;
import java.io.Serializable;
public class KafkaPartitioner implements Partitioner,Serializable {
  static int i = 0;
  //kafka 的分区函数返回值为分区索引，当前策略为轮询
  public int partition(Object key,int numPartitions) {
    i = (++i%numPartitions);
    return i;
  }
}
```

其中 KafkaPartitioner 必须实现 kafka.producer.Partitioner 的 partition 接口，其参数分别为消息的 Key 值，分区总数，返回值为分区的索引，当前策略仅仅是轮询，将消息均匀地分布在 Topic 所有的 Partition 上，具体实现可以根据不同的业务自己制定。

kafka.javaapi.producer.Producer 对外分别提供了发送单条消息的接口和发送多条消息的接口，每条消息是由 KeyedMessage 类表示，如下所示：

```
/* topic 指定消息的主题，key 指定消息的 key 值，partKey 指定消息的分区值，message 代表消息本身 */
case class KeyedMessage[K, V](
  val topic: String,
  val key: K,
  val partKey: Any,
  val message: V) {
  if(topic == null)
    throw new IllegalArgumentException("Topic cannot be null.")
  //如果传入 topic 和 message，则 key 和 partKey 为空
  def this(topic: String, message: V) = this(topic, null.asInstanceOf[K],
    null, message)
  //如果传入 topic，key 和 message，则 key 和 partKey 相等
  def this(topic: String, key: K, message: V) = this(topic, key, key, message)
  //分区 Key 优先取 partKey，如果不存在的话，则取 key
  def partitionKey = {
    if(partKey != null)
      partKey
    else if(hasKey)
      key
    else
      null
  }
  def hasKey = key != null
}
```

KeyedMessage 由 4 个元素组成：topic 指定消息的主题，key 指定消息的主键（可以参考

4.3.1.3 中有关日志压缩的说明），partKey 指定分区键，message 为消息的本身。KafkaPartitioner 会利用 KeyedMessage 中的 partKey 进行分区。

7.3 模块组成

由于 Kafka 是用 Scala 编写的，在 Java 代码中调用的 kafka.javaapi.producer.Producer 并不是 Kafka 内部真正的生产者实现类，kafka.javaapi.producer.Producer 只是对真正的生产者实现类 kafka.producer.Producer 进行了封装，其具体实现如下：

```
class Producer[K,V](private val underlying: kafka.producer.Producer[K,V]) {
  // 利用 ProducerConfig 初始化了 kafka.producer.Producer 对象
  def this(config: ProducerConfig) = this(new kafka.producer.Producer[K,V](config))
  def send(message: KeyedMessage[K,V]) {
    // 真正发送消息的是 kafka.producer.Producer 对象
    underlying.send(message)
  }
  def send(messages: java.util.List[KeyedMessage[K,V]]) {
    import collection.JavaConversions._
    // 真正发送消息的是 kafka.producer.Producer 对象
    underlying.send((messages: mutable.Buffer[KeyedMessage[K,V]]).toSeq: _*)
  }
  def close = underlying.close
}
```

可见在 Kafka 内部，生产者真正的实现类为 kafka.producer.Producer，客户端消息的发送最终是利用 kafka.producer.Producer 类提供的 send 接口发送出去的。

Producer 内部包含以下几个主要模块：

- ProducerSendThread：当 producer.type 配置为 async，则 ProducerSendThread 主要用于缓存客户端的 KeyedMessage，然后累计到 batch.num.messages 配置的数量之后或者间隔 queue.enqueue.timeout.ms 配置的时间（单位毫秒）还没有获取到新的客户端的 KeyedMessage，则调用 DefaultEventHandler 将 KeyedMessage 发送出去。
- ProducerPool：缓存客户端和各个 Broker Server 的通信，DefaultEventHandler 从 ProducerPool 中获取和某个 Broker Server 的通信对象 SyncProducer，然后通过 SyncProducer 将 KeyedMessage 发送给指定的 Broker Server。
- DefaultEventHandler：将 KeyedMessage 集合按照分区规则计算不同 Broker Server 所应该接收的部分 KeyedMessage，然后通过 SyncProducer 将 KeyedMessage 发送出去。在 DefaultEventHandler 模块内部提供了 SyncProducer 发送失败的重试机制和平滑扩容 Broker Server 的机制。

7.3.1 ProducerSendThread

当用户配置 producer.type 为 async 时，Producer 客户端会启动 ProducerSendThread 线

程，该线程负责从存放消息的 BlockingQueue 阻塞队列中获取消息，当超过一定数据量或者间隔一定时间没有获取到数据时，将当前累计取到的消息通过调用 DefaultEventHandler 模块发送出去，其具体实现如下：

```
class ProducerSendThread[K,V](
val threadName: String,
  val queue: BlockingQueue[KeyedMessage[K,V]],//存放消息的阻塞队列
  val handler: EventHandler[K,V],
  val queueTime: Long,//由 queue.buffering.max.ms 指定，默认为 5000ms
  val batchSize: Int,//由 queue.buffering.max.messages 指定，默认为 10000 条
  val clientId: String) extends Thread(threadName) with Logging with KafkaMetricsGroup {
  private def processEvents() {
    var lastSend = SystemTime.milliseconds
    var events = new ArrayBuffer[KeyedMessage[K,V]]
    var full: Boolean = false
    /* 从阻塞队列中拉取消息，其中超时时间设置为 (lastSend + queueTime) -
       SystemTime.milliseconds*/
    Stream.continually(
      queue.poll(scala.math.max(0, (lastSend + queueTime) - SystemTime.milliseconds),
      TimeUnit.MILLISECONDS)).takeWhile(item => if(item != null) item ne
        shutdownCommand
        else true).foreach {
          currentQueueItem =>
          //如果超时，则 currentQueueItem 为 null，expired 为 true
            val expired = currentQueueItem == null
            //计算累计的消息条数
            if(currentQueueItem != null) {
              events += currentQueueItem
            }
            //计算累计的消息条数是否超过 queue.buffering.max.messages 指定的值
            full = events.size >= batchSize
            //一旦累计过多消息或者获取消息超时，则发送消息
            if(full || expired) {
              ......
              //调用 DefaultEventHandler 将消息发送出去
              tryToHandle(events)
              //标记发送成功的时间
              lastSend = SystemTime.milliseconds
              //清空发送的消息
              events = new ArrayBuffer[KeyedMessage[K,V]]
            }
      }
    //发送最后一批消息
    tryToHandle(events)
    if(queue.size > 0)
      throw new IllegalQueueStateException(
        "Invalid queue state! After queue shutdown, %d remaining items in the queue"
        .format(queue.size))
  }
}
```

其中 queue.buffering.max.ms 指定客户端获取消息超时的时间，queue.buffering.max.messages 指定客户端缓存消息的最大个数，用户可以通过这两个参数来调节客户端异步发送的频率。

7.3.2 ProducerPool

ProducerPool 内部缓存了和不同 Broker Server 的通信链路，每个通信链路都由 SyncProducer 对象表示，该对象通过提供 doSend 接口将 ProducerRequest 和 TopicMetadataRequest 发送出去，其中 ProducerRequest 为生产者发送客户端消息的请求，TopicMetadataRequest 为生产者获取 Topic 元数据的请求。doSend 接口的具体实现如下：

```
class SyncProducer(val config: SyncProducerConfig) extends Logging {
  private def doSend(request: RequestOrResponse, readResponse: Boolean =
    true): Receive = {
    lock synchronized {
      //验证请求，调试目的
      verifyRequest(request)
      //如果当前没有和 Broker Server 建立链接，则建立链接，否则退出
      getOrMakeConnection()
      var response: Receive = null
      try {
        //调用阻塞通道将请求发送出去
        blockingChannel.send(request)
        if(readResponse)//设置是否需要读取响应
          //从阻塞通道获取响应
          response = blockingChannel.receive()
        else//忽略响应
          trace("Skipping reading response")
      } catch {
        //发送失败，则断开连接
        case e: java.io.IOException =>
          disconnect()
          throw e
        case e: Throwable => throw e
      }
      //返回响应
      response
    }
  }
}
```

在 SyncProducer 内部为什么发送的对象取名为 blockingChannel 呢？因为客户端发送请求利用的是 Socket 的阻塞模式，所以和 Broker Server 的通信链路取名为 blockingChannel，其内部就是打开一个 Socket 客户端，然后设置阻塞模式等参数，通过这个 Socket 客户端将请求真正发送出去，其实现如下：

```
class BlockingChannel(
```

```
  val host: String,
  val port: Int,
  val readBufferSize: Int, //保持 Socket 默认
  val writeBufferSize: Int, //由 send.buffer.bytes 指定，默认为 100*1024
  val readTimeoutMs: Int ) extends Logging {// 由 request.timeout.ms 指定，默认为 10000ms
  private var connected = false
  private var channel: SocketChannel = null
  private var readChannel: ReadableByteChannel = null
  private var writeChannel: GatheringByteChannel = null
  private val lock = new Object()
  private val connectTimeoutMs = readTimeoutMs
  def connect() = lock synchronized {
    if(!connected) {
      try {
        channel = SocketChannel.open()
        if(readBufferSize > 0)
          //设置读缓冲区大小
          channel.socket.setReceiveBufferSize(readBufferSize)
        if(writeBufferSize > 0)
          //设置写缓冲区大小
          channel.socket.setSendBufferSize(writeBufferSize)
        //配置阻塞模式
        channel.configureBlocking(true)
        //设置 Socket 超时时间
        channel.socket.setSoTimeout(readTimeoutMs)
        //开启 KeepAlive 模式
        channel.socket.setKeepAlive(true)
        /* 不启用 Nagle 算法，即客户端每发送一次数据，无论数据包的大小都会将这些数据发送出去 */
        channel.socket.setTcpNoDelay(true)
        channel.socket.connect(new InetSocketAddress(host, port), connectTimeoutMs)
        writeChannel = channel
        readChannel = Channels.newChannel(channel.socket().getInputStream)
        connected = true
      } catch {
        case e: Throwable => disconnect()
      }
    }
  }
}
```

ProducerPool 通过 BrokerId 将不同的 SyncProducer 一一映射，并且提供 updateProducer 接口来更新内部的 SyncProducer 连接池，即如下所示：

```
class ProducerPool(val config: ProducerConfig) extends Logging {
  //syncProducers 的 key 为 BrokerId, value 为 SyncProducer
  private val syncProducers = new HashMap[Int, SyncProducer]
  private val lock = new Object()
  def updateProducer(topicMetadata: Seq[TopicMetadata]) {
    val newBrokers = new collection.mutable.HashSet[Broker]
    //统计 Topic 所有分区的 Leader Replica
```

```
    topicMetadata.foreach(tmd => {
      tmd.partitionsMetadata.foreach(pmd => {
        if(pmd.leader.isDefined)
          newBrokers+=(pmd.leader.get)
      })
    })
    lock synchronized {
      newBrokers.foreach(b => {
        if(syncProducers.contains(b.id)){
          // 存在已有的 SyncProducer，则关闭之后新建
          syncProducers(b.id).close()
          syncProducers.put(b.id, ProducerPool.createSyncProducer(config, b))
        } else
          // 不存在 SyncProducer，则新建
          syncProducers.put(b.id, ProducerPool.createSyncProducer(config, b))
      })
    }
  }
}
```

7.3.3 DefaultEventHandler

DefaultEventHandler 决定了发送请求的具体逻辑，主要分以下几个步骤：

1）通过分区规则首先将 KeyedMessage 分组，不同的 KeyedMessage 落入不同的 Topic 的分区，然后按照分区的 Leader Replica 所在的 Broker Server 分组，每个 Broker Server 对应于不同的 KeyedMessage。

2）从 ProducerPool 中获取不同的 Broker Server 对应的 SyncProducer 对象，然后通过 SyncProducer 对象将 KeyedMessage 发送出去

除此之外，DefaultEventHandler 还提供发送失败的重试机制和平滑扩容 Broker Server 的机制。

首先来看 DefaultEventHandler 处理的主要逻辑，如下所示：

```
def handle(events: Seq[KeyedMessage[K,V]]) {
  // 序列化 events
  val serializedData = serialize(events)
  var outstandingProduceRequests = serializedData
  // 设置失败重试次数，由 message.send.max.retries 配置，默认为 3
  var remainingRetries = config.messageSendMaxRetries + 1
  // 获取客户端发送消息的 correlationId，相同的 correlationId 表示对应的请求和响应
  val correlationIdStart = correlationId.get()
  // 重试次数没有达到且还有待发送的数据
  while (remainingRetries > 0 && outstandingProduceRequests.size > 0) {
    // 获取待发送消息的 Topic 集合
    topicMetadataToRefresh ++= outstandingProduceRequests.map(_.topic)
    /* 如果长时间没有刷新 Topic 元数据，则主动刷新，其中时间间隔由
       topic.metadata.refresh.interval.ms 配置，默认为 600000ms*/
    if (topicMetadataRefreshInterval >= 0 &&
```

```
        SystemTime.milliseconds - lastTopicMetadataRefreshTime >
          topicMetadataRefreshInterval) {
        /* 如果 Topic 在中途新增 Partition 到其他 Broker Server 上，则此时会被发现，并在
          ProducerPool 中缓存对应的 SyncProducer*/
        Utils.swallowError(
          brokerPartitionInfo.updateInfo(
            topicMetadataToRefresh.toSet,
            correlationId.getAndIncrement))
        /* 清理 sendPartitionPerTopicCache，里面保存如果没有配置 Topic 分区函数情况下消息
          发往的默认分区 */
        sendPartitionPerTopicCache.clear()
        // 清理待刷新的 Topic 缓存
        topicMetadataToRefresh.clear
        // 记录刷新时间
        lastTopicMetadataRefreshTime = SystemTime.milliseconds
      }
      // 发送消息，返回发送失败的消息，需要重新发送
      outstandingProduceRequests = dispatchSerializedData(outstandingProduceRequests)
      if (outstandingProduceRequests.size > 0) {
        // 发送失败，则 sleep 一段时间，由 retry.backoff.ms 配置，默认为 100ms
        Thread.sleep(config.retryBackoffMs)
        /* 发送失败可能是 Topic 的元数据信息发生变化，则此时需要更新 Topic 的元数据信息，并且同
          时更新 ProducerPool*/
        Utils.swallowError(brokerPartitionInfo.updateInfo(
          outstandingProduceRequests.map(_.topic).toSet,
          correlationId.getAndIncrement))
        /* 清理 sendPartitionPerTopicCache，里面保存如果没有配置 Topic 分区函数情况下消息
          发往的默认分区 */
        sendPartitionPerTopicCache.clear()
        // 重试次数递减
        remainingRetries -= 1
      }
    }
    if(outstandingProduceRequests.size > 0) {
      // 超过一定的重试次数之后还是没有发生成功，则抛出异常
      throw new FailedToSendMessageException(
        "Failed to send messages after " + config.messageSendMaxRetries + " tries.",
        null)
    }
  }
```

DefaultEventHandler 为什么间隔一定的时间需要刷新 Topic 的元数据信息，并同时更新 ProducerPool 呢？这是因为当 Kafka 集群新增 Broker 节点时，并且同时 Topic 新增分区到该节点时，ProducerPool 内部缓存的还是该 Topic 在集群新增节点之前的 Broker Server 通信链路集合。为了使 ProducerPool 能够感知 Topic 元数据信息的变化，应对 Kafka 集群新增 Broker 节点的场景，需要有一种机制能够使 Producer 实现平滑的扩容，此时需要利用 Producer 配置文件中的 Broker 列表，间隔一定时间之后，随机向某个 Broker Server 发送 TopicMetadataRequest 请求，获取 Topic 的元数据信息，然后更新 ProducerPool。这就是

DefaultEventHandler 提供的平滑扩容 Broker Server 的机制。

除此之外，如果当 Topic 的 Partition 的 Leader Replica 发生切换，则发送给切换前的 Leader Replica 必然会导致失败，那么 DefaultEventHandler 在消息发送失败的时候，需要重新更新 Topic 的元数据，并且更新 ProducerPool。

DefaultEventHandler 更新 Topic 的元数据信息流程如下：

```
class BrokerPartitionInfo(producerConfig: ProducerConfig,
                          producerPool: ProducerPool,
                          topicPartitionInfo: HashMap[String, TopicMetadata])
                          extends Logging {
  val brokerList = producerConfig.brokerList
  val brokers = ClientUtils.parseBrokerList(brokerList)
  def updateInfo(topics: Set[String], correlationId: Int) {
    var topicsMetadata: Seq[TopicMetadata] = Nil
    /* 随机选择 metadata.broker.list 指定的 Broker 列表中的 Broker Serevr 发送
       TopicMetadataRequest 请求 */
    val topicMetadataResponse = ClientUtils.fetchTopicMetadata(
      topics,
      brokers,
      producerConfig,
      correlationId)
    topicsMetadata = topicMetadataResponse.topicsMetadata
    topicsMetadata.foreach(tmd =>{
      if(tmd.errorCode == ErrorMapping.NoError) {
        // 更新 TopicMetadata
        topicPartitionInfo.put(tmd.topic, tmd)
      } else
        ......
    })
    // 刷新 ProducerPool
    producerPool.updateProducer(topicsMetadata)
  }
}
```

从上述流程可以看到：

1）生产者更新 Topic 元数据利用的是 TopicMetadataRequest 请求来获取 Topic 的元数据。

2）更新完 Topic 元数据之后还需要更新 ProducerPool，因为里面包含了连接不同 Broker Server 的通信链路。

DefaultEventHandler 的主流程在正常情况下会进入 dispatchSerializedData 流程，在 dispatchSerializedData 流程中首先会把 KeyedMessage 按照 Broker Server 分组，如果 KeyedMessage 对应的 Partition 的 Leader Replica 所在的 Broker Server 相同的话，则这些 KeyedMessage 会分在同一组，其次会把同一组的 KeyedMessage 利用 ProducerPool 中缓存的 SyncProducer 发送到对应的 Broker Server，其具体实现如下：

```
// 输入待发送的 KeyedMessage，返回发送失败的 KeyedMessage
```

```
private def dispatchSerializedData(messages: Seq[KeyedMessage[K,Message]])
  : Seq[KeyedMessage[K, Message]] = {
  /* 将消息分组，其 partitionedDataOpt 类型为 Map[BrokerId,Map[TopicAndPartition,Seq
     [Message]]] 将发往相同 Broker Server 的消息分为一组 */
  val partitionedDataOpt = partitionAndCollate(messages)
  partitionedDataOpt match {
    case Some(partitionedData) =>
      val failedProduceRequests = new ArrayBuffer[KeyedMessage[K,Message]]
      try {
        for ((brokerid, messagesPerBrokerMap) <- partitionedData) {
          /* 聚合发往同一个 Broker Server 的消息集合，即将不同 TopicAndPartition 的
             Seq[Message] 聚合在一起 */
          val messageSetPerBroker = groupMessagesToSet(messagesPerBrokerMap)
          // 向 brokerid 发送 messageSetPerBroker(消息集合)，返回发送失败的 TopicAndPartition
          val failedTopicPartitions = send(brokerid, messageSetPerBroker)
          failedTopicPartitions.foreach(topicPartition => {
            messagesPerBrokerMap.get(topicPartition) match {
              // 根据 topicPartition 返回发送失败的消息集合
              case Some(data) => failedProduceRequests.appendAll(data)
              case None =>
            }
          })
        }
      } catch {
        case t: Throwable => error("Failed to send messages", t)
      }
      // 返回发送失败的消息集合
      failedProduceRequests
    case None =>
      // messages 分组失败，即无法发送消息，则全部返回
      messages
  }
}
```

可见 dispatchSerializedData 的关键是如何针对单条 KeyedMessage 进行分组，一旦分组成功，即找到了目的 Broker Server，则接下来就是利用 SyncProducer 发送出去。DefaultEventHandler 内部提供了 getPartition 函数，输入 Topic、partKey、Partition List，输出 Partition 索引，其具体实现如下所示：

```
private def getPartition(
  topic: String,
  key: Any,
  topicPartitionList: Seq[PartitionAndLeader]): Int = {
  // 计算 Topic 分区个数
  val numPartitions = topicPartitionList.size
  // 分区个数必须大于 0，否则无法路由
  if(numPartitions <= 0)
```

```
        throw new UnknownTopicOrPartitionException("Topic " + topic + " doesn't exist")
      val partition =
      if(key == null) {
        // 分区键为空，则从 cache 中获取，每个 Topic 只发往一个分区
        val id = sendPartitionPerTopicCache.get(topic)
        id match {
          // 从 cache 中获取到分区索引
          case Some(partitionId) =>
            partitionId
          // 从 cache 中获取不到分区索引，需要重新计算
          case None =>
            // 筛选出 Leader Replica 存在的分区列表
            val availablePartitions = topicPartitionList.filter(_.leaderBrokerIdOpt.
              isDefined)
            if (availablePartitions.isEmpty)
              throw new LeaderNotAvailableException("No leader for any partition
                in topic " + topic)
            // 针对 availablePartitions 利用随机值取模
            val index = Utils.abs(Random.nextInt) % availablePartitions.size
            // 获取对应的分区索引
            val partitionId = availablePartitions(index).partitionId
            // 更新 cache
            sendPartitionPerTopicCache.put(topic, partitionId)
            // 返回分区索引
            partitionId
        }
      } else
        /* 分区键不为空，则调用分区函数进行分区，此分区函数用户可以自定义，或者利用默认的分区函数
          DefaultPartitioner*/
        partitioner.partition(key, numPartitions)
      // 校验分区索引的有效性
      if(partition < 0 || partition >= numPartitions)
        throw new UnknownTopicOrPartitionException("Invalid partition id: " +
          partition + " for topic " + topic + "; Valid values are in the inclusive
          range of [0, " + (numPartitions-1) + "]")
      // 返回分区索引
      partition
    }
```

可见如果没有传入分区键，则首先会随机选择某个有效的分区，然后固定发往这个有效的分区，并且当触发刷新 Topic 元数据的时候，会清除之前的映射关系，重新计算某个分区，然后向其发生数据，因此在客户端的现象就是：消息一段时间集中发往某个分区，然后过一段时间又集中发往另外一个分区。如果用户传入分区键，则会调用分区函数进行分区，分区函数的第二个参数为分区个数。并且如果平滑扩容 Broker Server，则在重新更新 Topic 元数据时会刷新该 Topic 的分区个数，此时新加入节点的 Broker Server 可以接收到生产者的消息。

除此之外需要提醒大家的是：Kafka 提供的默认分区函数 DefaultPartitioner 采用的是 Hash 分区。其具体实现如下：

```
class DefaultPartitioner(props: VerifiableProperties = null) extends Partitioner {
  private val random = new java.util.Random
  def partition(key: Any, numPartitions: Int): Int = {
    // 针对分区键进行 Hash，相同的分区键发往相同的分区
    Utils.abs(key.hashCode) % numPartitions
  }
}
```

7.4 发送模式

生产者发送消息分为两种模式：1）同步模式；2）异步模式。在同步模式下发送消息，消息立刻发送到指定的 Broker Server；在异步模式下发送消息，消息并不是立刻发送到指定的 Broker Server，而是缓存在客户端消息队列缓冲区，并且等待消息队列缓冲区是否达到一定数量或者间隔一定的时间之后才把消息发送出去。因此接下来将针对这两种模式，并且结合 7.3 节描述的模块组成，详细阐述不同模式下消息发送的原理。

7.4.1 同步模式

当 producer.type 配置为 sync 时，生产者采用同步方式发生消息，即生产者接收到多少消息，则立刻发送多少消息，并不会在客户端缓存消息，其发送流程如图 7-2 所示。

在同步模式流程中存在三处需要更新 Topic 元数据信息的场景：1）发送前的定时更新机制；2）发送过程中针对消息进行分组，如果发现该消息的 Topic 元数据信息不存在，则需要更新；3）发送失败时的更新机制。当消息发送失败的时候，会进入重试流程，首先更新 Topic 的元数据信息，然后再次发送，如果发生成功，则直接退出。

7.4.2 异步模式

当 producer.type 配置为 async 时，生产者采用异步方式发生消息，即生产者会缓存一定数量的消息或者达到一定的超时时间才会把接收到的消息发送出去，其发送流程如图 7-3 所示。

异步方式发送消息与同步方式发送消息相比较，新增了一个异步发送线程，负责缓存客户端的消息，当客户端累积到一定的数据量或者间隔一定的时间之后，才把当前缓存的消息发送出去。因此客户端并不像同步方式发送消息那样直接发送消息，而是把消息存放进消息队列缓冲区。假如在发送过程中客户端由于异常退出，则很可能会丢失那些还在缓冲区的消息，因此针对异步方式发送消息，客户端需要针对异常退出的情况进行特殊处理。

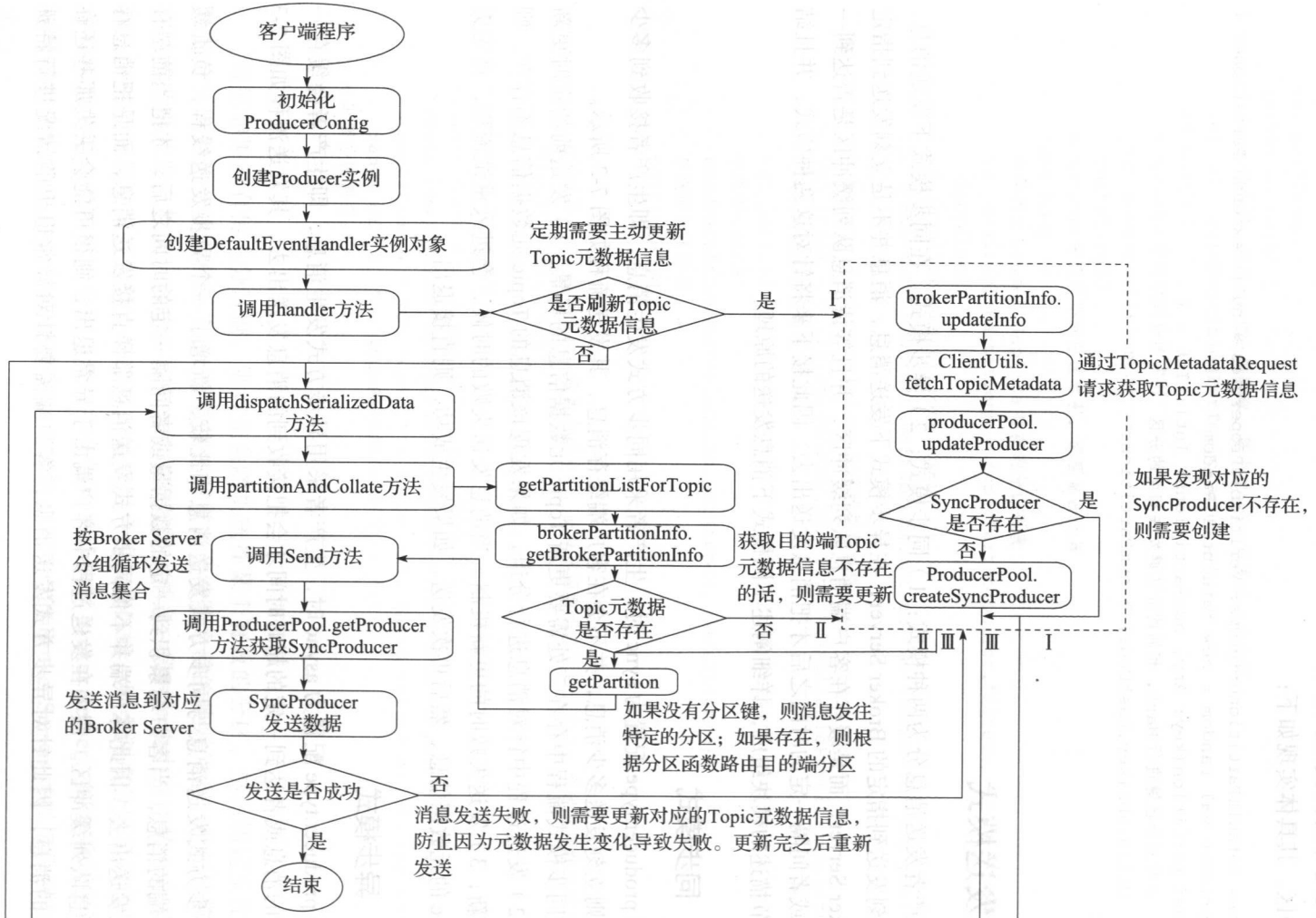


图 7-2　同步方式发送消息流程

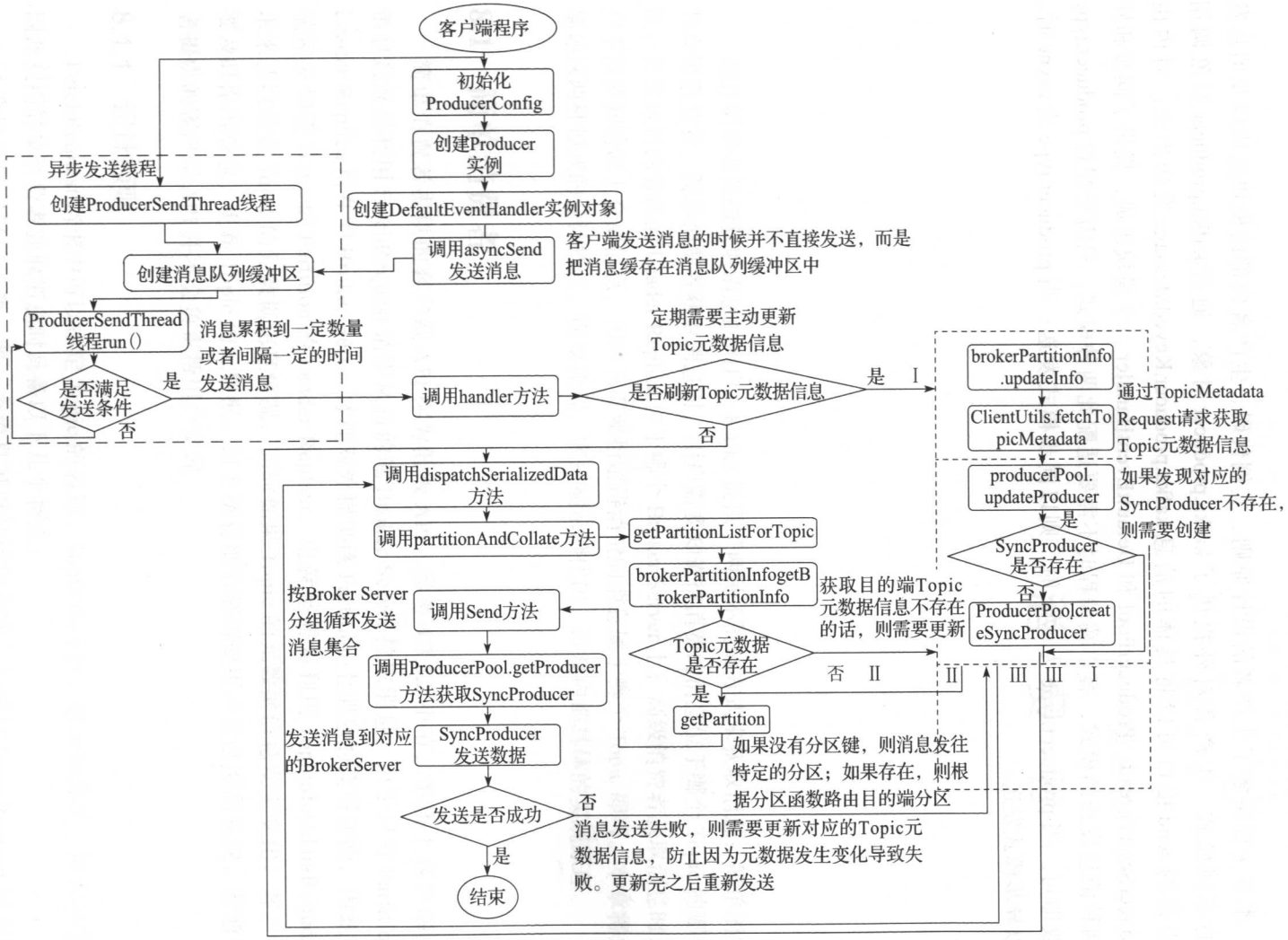
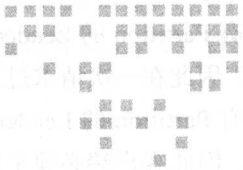


图 7-3　异步方式发送消息流程

7.5 本章小结

本章主要讲解了生产者的设计原则、实例代码、生产者内部的模块组成以及消息发送的不同模式。生产者对外提供了 kafka.producer 对象，通过 kafka.producer 对象调用其提供的 send 接口可以将具体的消息 kafka.producer.KeyedMessage 发送出去；对内由 ProducerSendThread、ProducerPool 和 DefaultEventHandler 三个模块组成，提供了同步和异步两种消息发送的模式，客户端编程时不需要适配此两种模式，只需要配置 producer.type 参数即可。当 producer.type 为 sync 时，则以同步模式发送；当 producer.type 为 async 时，则以异步模式发送。



第 8 章 Chapter 8

消 费 者

消息消费者是指获取消息的用户。Kafka 提供了两种不同的方式来获取消息：简单消费者和高级消费者。简单消费者获取消息时，用户需要知道待消费的消息位于哪个 Topic 的哪个分区，并且该目的分区的 Leader Replica 位于哪个 Broker Server 上；高级消费者获取消息时，用户不需要知道以上全部信息，用户只需要指定待消费的消息属于哪个 Topic 即可。本章将分别描述这两种模式的设计特点、简要流程、客户端示例代码，以及内部具体的实现原理。

8.1 简单消费者

简单消费者提供的客户端 API 称为低级 API，参考 4.3.5.2 小节，本质上客户端获取消息最终是利用 FetchRequest 请求从目的端 Broker Server 拉取消息。由于只有 Partition 的 Leader Replica 所在的 Broker Server 才能对外提供该 Partition 上消息的读写功能，因此客户端需要知道 Topic 的 Partition 的 Leader Replica，也就是需要利用 TopicMetadataRequest 请求来获取指定 Topic 的元数据信息情况，同时如果 Topic 的元数据信息发生变化，客户端需要及时作出响应，更新 Topic 的元数据。以上所有细节都需要用户自己编码实现，普通消费者提供的客户端 API 并不会负责帮用户实现。

8.1.1 设计原则

FetchRequest 请求中可以指定 Topic 的名称、Topic 的分区、起始偏移量、最大字节数。因此利用简单消费者获取消息时具备以下几个特点：

- 消息可以重复被消费，即一个消息可以被读取多次，只要其 FetchRequest 里面的参

数相同即可。

- 由于只有 Partition 的 Leader Replica 所在的 Broker Server 对外提供该 Partition 的消息读写，因此在一次请求过程中只能获取 Topic 的部分 Partition 数据。
- 由于只有 Partition 的 Leader Replica 所在的 Broker Server 对外提供该 Partition 的消息读写，因此客户端必须主动找出待消费 Partition 的 Leader Replica。
- 为了避免重复消费消息，客户端需要主动跟踪 Partition 的偏移量，无论该偏移量最终持久化至客户端提供的外部存储介质上还是 Kafka 内部。
- 由于消费过程中 Topic 的元数据信息可能会发生变化，则客户端必须及时处理 Topic 的元数据信息改变的情况。

8.1.2 消费者流程

一次成功的客户端消费过程如图 8-1 所示。

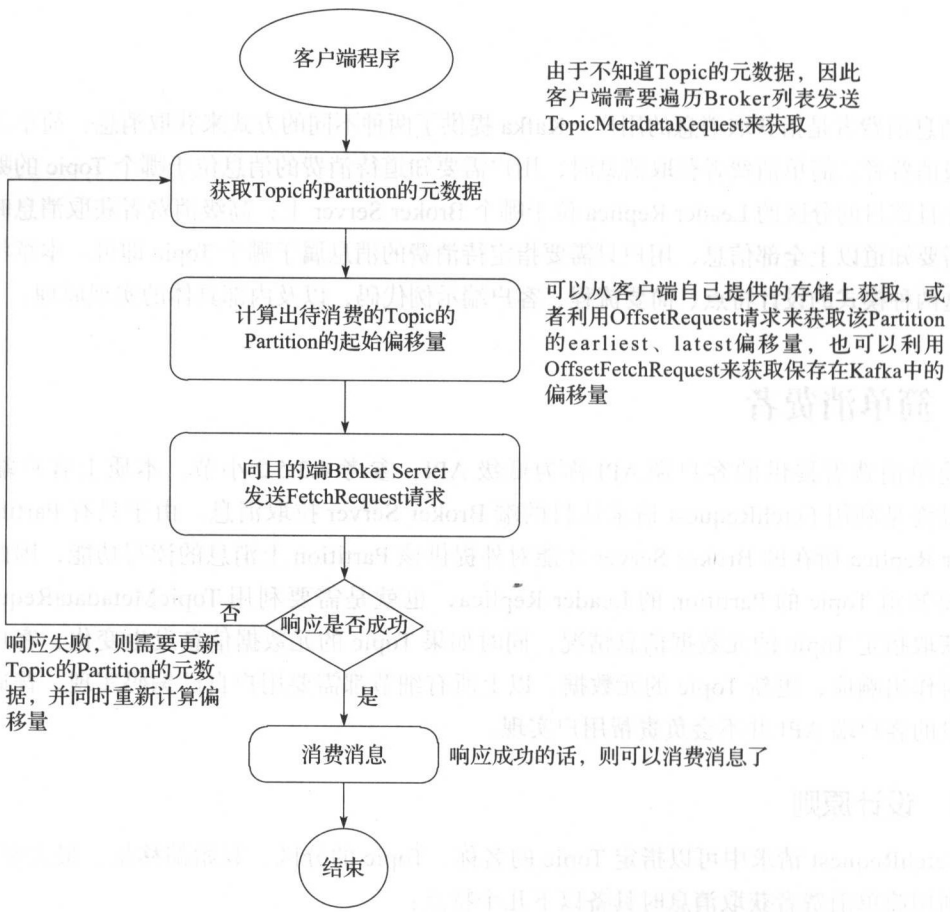


图 8-1 客户端简单消费者流程

图 8-1 为单次成功消费的流程，如果要持续不断地消费，则其每次消费的偏移量需要利用某种机制保存起来：1）保存至客户端自己提供的外部存储介质上；2）利用普通消费者提供的 commitOffsets 接口将偏移量保存至 Kafka 之中。

8.1.3 示例代码

按照上节所示客户端简单消费者流程，客户端的示例代码如下：

```
public class KafkaSimpleConsumer {
  //保存 Partition 的 Replica 所在的 Broker 列表
  private List<String> replicaBrokers = new ArrayList<String>();
  public KafkaSimpleConsumer() {
    replicaBrokers = new ArrayList<String>();
  }
  public static void main(String args[]) {
    KafkaSimpleConsumer example = new KafkaSimpleConsumer();
    //重试次数
    ong maxReads = Long.parseLong("3");
    //待消费的 Topic 名称
    String topic = "mytopic";
    //分区索引
    int partition = Integer.parseInt("0");
    /* 获取 Topic 的 Partition 的元数据所需要的 Broker 列表，最好包含 Kafka 集群所有的 Broker
       Server*/
    List<String> seeds = new ArrayList<String>();
    seeds.add("172.23.8.160");
    seeds.add("172.23.8.161");
    seeds.add("172.23.8.162");
    //端口号
    int port = Integer.parseInt("9092");
    try {
      example.run(maxReads, topic, partition, seeds, port);
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
  public void run(long maxReads,
                  String topic,
                  int partition,
                  List<String> seedBrokers,
                  int port) throws Exception {
    //获取 Topic 的 Partition 的元数据
    PartitionMetadata metadata = findLeader(seedBrokers, port, topic, partition);
    //无法获取 Topic 的 Partition 的元数据，则退出。
    if (metadata == null) {
      return;
    }
    //如果此时 Partition 的 leader 不存在，则退出
    if (metadata.leader() == null) {
```

```
        return;
    }
    // 获取 Leader Replica 所在的 Broker
    String leadBroker = metadata.leader().host();
    String clientName = "Client_" + topic + "_" + partition;
    /* 利用 SimpleConsumer 来获取特定 Partition 的数据，其主要参数为 Leader Replica 所在
       的 Broker、端口号 */
    SimpleConsumer consumer = new SimpleConsumer(
      leadBroker,
      port,
      100000,
      64 * 1024,
      clientName);
    /*
      计算偏移量。可以通过多种方式计算偏移量：
      1）利用 OffsetRequest 请求查询该 Topic 的 Partition 的 earliest、latest 偏移量或者早于某
      个时间的偏移量
      2）利用 OffsetFetchRequest 请求查询客户端之前提交到 Kafka 中的偏移量
      3）客户端每次消费记录偏移量至外部存储介质上，再次消费的时候从外部存储介质加载
      此处采用第一种方式，查询早于当前时间的偏移量
     */
    long readOffset = getLastOffset(
      consumer,
      topic,
      partition,
      kafka.api.OffsetRequest.EarliestTime(),
      clientName);
    int numErrors = 0;
    // 重试次数大于 0
    while (true) {
      // 消费失败的时候，需要重新建立 SimpleConsumer
      if (consumer == null) {
        consumer = new SimpleConsumer(leadBroker, port, 100000, 64 * 1024,
          clientName);
      }
      // 组装 FetchRequest，主要指定 topic、partition、readOffset
      FetchRequest req = new FetchRequestBuilder().clientId(clientName).addFetch(
        topic,
        partition,
        readOffset,
        100000).build();
      // 利用 SimpleConsumer 发送 FetchRequest 请求，获取响应
      FetchResponse fetchResponse = consumer.fetch(req);
      if (fetchResponse.hasError()) {
        numErrors++;
        // fetch 失败，则获取错误码
        short code = fetchResponse.errorCode(topic, partition);
        // 重试次数大于 3，则退出
        if (numErrors > 3)
          break;
```

```
        if (code == ErrorMapping.OffsetOutOfRangeCode()) {
          //偏移量越界的话，则重新计算偏移量
          readOffset = getLastOffset(
            consumer,
            topic,
            partition,
            kafka.api.OffsetRequest.LatestTime(),
            clientName);
          continue;
        }
        //关闭 SimpleConsumer
        consumer.close();
          consumer = null;
          //重新查找 Topic 的 Partition 的 Leader Replica 所在的 Broker Server
          leadBroker = findNewLeader(leadBroker, topic, partition, port);
          continue;
        }
        numErrors = 0;
        long numRead = 0;
        //消息获取成功，则消费消息
        for (MessageAndOffset messageAndOffset : fetchResponse.messageSet(topic,
          partition)) {
          long currentOffset = messageAndOffset.offset();
          if (currentOffset < readOffset) {
            //剔除小于设置偏移量的消息
            continue;
          }
          /*
            更新当前偏移量，存在多种方式保存偏移量：
            1）保存在外部存储介质上，客户端重启之后不丢失
            2）利用 OffsetCommitRequest 提交至 Kafka，客户端重启之后不丢失
            3）保存在客户端内存中，客户端重启之后丢失
            此处采用第三种方式
           */
          readOffset = messageAndOffset.nextOffset();
          ByteBuffer payload = messageAndOffset.message().payload();
          byte[] bytes = new byte[payload.limit()];
          //获取当前消息
          payload.get(bytes);
          numRead++;
        }
        if (numRead == 0) {
          //消费没有获取到，则休眠一段时间
          try {
            Thread.sleep(1000);
          } catch (InterruptedException ie) {

          }
        }
      }
```

```
        if (consumer != null)
          consumer.close();
      }
    }
  }
```

在 run 的主流程中获取消息的步骤如下：

步骤 1 获取 Topic 的 Partition 的元数据。

步骤 2 计算当前需要消费的偏移量。

步骤 3 向 Topic 的 Partition 的 Leader Replica 所在的 Broker Server 发送 FetchRequest，如果失败，则返回步骤 1。

步骤 4 消费消息。

步骤 5 更新偏移量。

其中步骤 1 通过 findLeader 获取 Topic 的 Partition 的元数据，其具体实现如下：

```
PartitionMetadata findLeader(
  List<String> seedBrokers,
  int port,
  String topic,
  int partition) {
  PartitionMetadata returnMetaData = null;
  //遍历初始的 Broker Server 列表
  for (String seed : seedBrokers) {
    SimpleConsumer consumer = null;
    try {
      //针对 Broker Server 新建一个 SimpleConsumer
      consumer = new SimpleConsumer(seed, port, 100000, 64 * 1024, "leaderLookup");
      List<String> topics = Collections.singletonList(topic);
      //组装 TopicMetadataRequest
      TopicMetadataRequest req = new TopicMetadataRequest(topics);
      //利用 SimpleConsumer 发送 TopicMetadataRequest
      kafka.javaapi.TopicMetadataResponse resp = consumer.send(req);
      List<TopicMetadata> metaData = resp.topicsMetadata();
      for (TopicMetadata item : metaData) {
        for (PartitionMetadata part : item.partitionsMetadata()) {
          if (part.partitionId() == partition) {
            //Partition 的元数据找到，则退出
            returnMetaData = part;
            break;
          }
        }
      }
    } catch (Exception e) {
    } finally {
      if (consumer != null)
        consumer.close();
    }
```

```
    }
    // 更新 replicaBrokers，将其设置为 Partition 的 Replica 列表所在的 Broker Server
    if (returnMetaData != null) {
      replicaBrokers.clear();
      for (kafka.cluster.Broker replica : returnMetaData.replicas()) {
        replicaBrokers.add(replica.host());
      }
    }
    // Topic 的 Partition 的元数据
    return returnMetaData;
  }
```

当在步骤3中获取消息失败时，需要更新Topic的Partition的元数据，此时遍历的Broker Server列表变为之前该Parition的Replica列表所在的Broke Server列表，即通过findNewLeader来获取Topic的Partition的元数据，其具体实现代码如下：

```
String findNewLeader(
    String oldLeader,
    String topic,
    int partition,
    int port) throws Exception {
    // 重试 3 次
    for (int i = 0; i < 3; i++) {
      boolean goToSleep = false;
      // 通过之前的 findLeader 进行查找，其中 seedBrokers 为 replicaBrokers
      PartitionMetadata metadata = findLeader(replicaBrokers, port, topic, partition);
      if (metadata == null) {
        // 查不到则休眠
        goToSleep = true;
      } else if (metadata.leader() == null) {
        // 查不到则休眠
        goToSleep = true;
      } else if (oldLeader.equalsIgnoreCase(metadata.leader().host()) && i == 0) {
        /* 如果第一次获取时 Partition 的 Leader Replica 还是保持不变，则可能是 Zookeeper 挂起了，
           需要休眠重试 */
        goToSleep = true;
      } else {
        // 找到则返回 Partition 的 Leader Replica 所在的 Broker Server
        return metadata.leader().host();
      }
      if (goToSleep) {
        // 休眠 1s
        try {
          Thread.sleep(1000);
        } catch (InterruptedException ie) {
        }
      }
    }
    throw new Exception("Unable to find new leader after Broker failure. Exiting");
}
```

在步骤 2 中通过 getLastOffset 来获取小于当前时间的偏移量，其具体实现如下：

```
public long getLastOffset(
  SimpleConsumer consumer,
  String topic,
  int partition,
  long whichTime,
  String clientName) {
  //组装 TopicAndPartition
  TopicAndPartition topicAndPartition = new TopicAndPartition(topic, partition);
  Map<TopicAndPartition, PartitionOffsetRequestInfo> requestInfo =
    new HashMap<TopicAndPartition, PartitionOffsetRequestInfo>();
  //设置偏移量小于 whichTime，且只需要获取 1 条偏移量记录即可 requestInfo.put(topicAndPartition,
    new PartitionOffsetRequestInfo(whichTime, 1));
  //组装 OffsetRequest
  OffsetRequest request =
    new OffsetRequest(requestInfo,
                      kafka.api.OffsetRequest.CurrentVersion(),
                      clientName);
  //通过 SimpleConsumer 发送 OffsetRequest
  OffsetResponse response = consumer.getOffsetsBefore(request);
  if (response.hasError()) {
    //错误，则返回 0
    return 0;
    }
  long[] offsets = response.offsets(topic, partition);
  //返回偏移量
  return offsets[0];
}
```

可见，如果是简单消费者模式的话，则客户端需要做很多事情：查找元数据，设置偏移量，保存偏移量，等等。但是在简单消费者模式下，消费数据会提供给用户更大的灵活性去控制整个消费的过程。

8.1.4 原理解析

在上节中所有请求的发送都是利用 SimpleConsumer 对象实现的，由于 Kafka 是由 Scala 编写的，在 Java 代码中调用的 kafka.javaapi.consumer. SimpleConsumer 并不是 Kafka 内部真正的简单消费者实现类，而是对真正的消费者实现类 kafka.consumer.SimpleConsumer 进行了封装，其具体实现如下：

```
class SimpleConsumer(val host: String,
                     val port: Int,
                     val soTimeout: Int,
                     val bufferSize: Int,
                     val clientId: String) {
  /* 内部真正的实现类为 kafka.consumer.SimpleConsumer，并且任何请求的发送最终是利用
     underlying 来实现的 */
```

```
  private val underlying = new kafka.consumer.SimpleConsumer(
    host,
    port,
    soTimeout,
    bufferSize,
    clientId)
  def fetch(request: kafka.api.FetchRequest): FetchResponse = {
    import kafka.javaapi.Implicits._
    underlying.fetch(request)
  }
  def fetch(request: kafka.javaapi.FetchRequest): FetchResponse = {
    fetch(request.underlying)
  }
  def send(request: kafka.javaapi.TopicMetadataRequest): kafka.javaapi.
    TopicMetadataResponse = {
    import kafka.javaapi.Implicits._
    underlying.send(request.underlying)
  }
  def getOffsetsBefore(request: OffsetRequest): kafka.javaapi.OffsetResponse = {
    import kafka.javaapi.Implicits._
    underlying.getOffsetsBefore(request.underlying)
  }
  def commitOffsets(request: kafka.javaapi.OffsetCommitRequest): kafka.javaapi.
    OffsetCommitResponse = {
      import kafka.javaapi.Implicits._
      underlying.commitOffsets(request.underlying)
  }
  def fetchOffsets(request: kafka.javaapi.OffsetFetchRequest): kafka.javaapi.
    OffsetFetchResponse = {
    import kafka.javaapi.Implicits._
    underlying.fetchOffsets(request.underlying)
  }
  def close() {
    underlying.close
  }
}
```

那么 kafka.consumer.SimpleConsumer 又是如何实现发送请求的呢？且看如下代码：

```
class SimpleConsumer(val host: String,
                     val port: Int,
                     val soTimeout: Int,
                     val bufferSize: Int,
                     val clientId: String) extends Logging {
  private val lock = new Object()
  //针对 host 和 port 建立阻塞的通信链路
  private val blockingChannel = new BlockingChannel(
    host,
    port,
    bufferSize,
    BlockingChannel.UseDefaultBufferSize, soTimeout)
```

```
  private def sendRequest(request: RequestOrResponse): Receive = {
    lock synchronized {
      var response: Receive = null
      try {
        //链接 Broker Server
        getOrMakeConnection()
        //利用该 blockingChannel 发送请求
        blockingChannel.send(request)
        //阻塞等待从 blockingChannel 获取响应
        response = blockingChannel.receive()
      } catch {
        //失败重连，重发
        case e : Throwable =>
          try {
            reconnect()
            blockingChannel.send(request)
            response = blockingChannel.receive()
          } catch {
            case e: Throwable =>
              disconnect()
              throw e
          }
      }
      response
    }
  }
}
```

回想 7.3.2 小节，客户端无论生产消息还是消费消息，最终都是通过与目的端 Broker Server 建立通信链路，并且以阻塞模式运行，然后通过该条链路将不同的请求发送出去。

8.2 高级消费者

高级消费者提供的客户端 API 称为高级 API。高级消费者是相对于简单消费者而言的，高级消费者屏蔽了使用简单消费者所需要做的许多额外工作，以 Consumer Group（消费组）的形式来管理消息的消费，以 Stream（流）的形式来提供具体消息的读取。客户端在消费消息的时候只需要知道消息所属的 Topic 即可，而不需要知道该 Topic 的具体元数据信息，因此高级 API 屏蔽了许多底层实现细节，使客户端只需要专注于具体的 Topic 即可。

8.2.1 设计原则

因为高级 API 通过 Consumer Group（消费组）的形式来管理消息的消费，只需要专注于具体的 Topic 而不需要知道其他额外的信息，因此高级消费者获取消息时具备以下几个特点：

- 客户端以某个 Consumer Group 消费消息时，消费过的消息无法再次消费，除非更换另外一个 Consumer Group 来消费消息。

- 为了记录 Consumer Group 消费消息的具体详情，其内部会开启一个提交偏移量的定时任务，将 Consumer Group 消费的详情提交至 Zookeeper 或者 Kafka 中保存。
- 客户端通过 Stream（流）的形式消费消息，Stream 是指来自若干个 Broker Server 上的若干个 Partition 的消息。客户端需要正确设置 Stream 的个数，并且应该针对每个 Sream 开启一个线程进行消息的消费。总而言之，一个 Stream 代表了多个 Partition 消息的聚合，但是每一个 Partition 只能映射到一个 Stream。
- Stream 和 Partition 的关系：
  a）如果 Stream 比 Partition 多，是浪费，因为 Kafka 的设计在一个 Partition 上是不允许并发的，所以 Stream 数不要大于 Partition 数，多余的 Stream 不会获取到消息。
  b）如果 Stream 比 Partition 少，一个 Stream 会对应于多个 Partition，这里需要合理分配 Stream 数和 Partition 数，否则会导致 Partition 之间的数据消费不均匀。
  c）如果 Stream 和 Partition 一样，则一个 Stream 会对应于一个 Partition。
  d）如果 Stream 从多个 Partition 读数据，则不保证数据间的顺序性，Kafka 只保证在一个 Partition 上数据是有序的，但多个 Partition 之间的数据，根据你读的顺序会有不同。
- 同一个 Consumer Group 的不同客户端之间的 Stream 会自动实现负载均衡。当 Topic 的分区发生变化，或者 Stream 的个数发生变化时，Kafka 会自动更新不同的 Stream 和 Partition 之间的映射关系，尽量做到 Stream 消费消息时集群内部是负载均衡的。

8.2.2 消费者流程

高级消费者 API 比低级消费者 API 消费消息更加简单和方便，只需要设置好 Stream 和 Partition 的关系，并开启若干个线程消费 Stream 即可，其客户端消费消息的过程如图 8-2 所示。

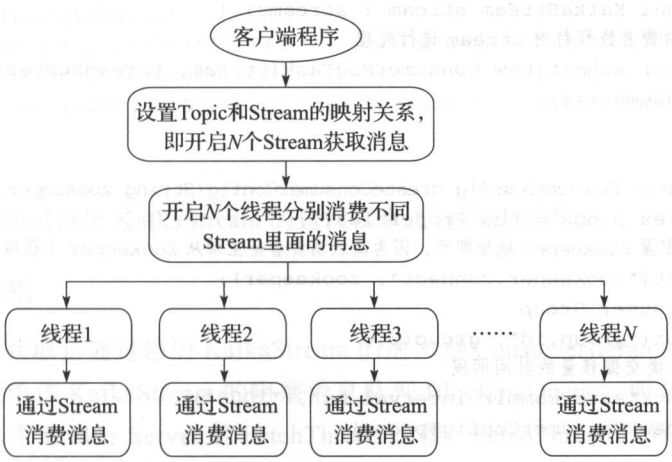


图 8-2 客户端高级消费者流程

8.2.3 示例代码

按照上节所示的客户端高级消费者流程，客户端的示例代码如下：

```
public class KafkaHighConsumer {
  private final ConsumerConnector consumer;
  private final String topic;
  private ExecutorService executor;
  public KafkaHighConsumer(String zookeeper, String groupId, String topic) {
    //利用 kafka.consumer.Consumer 创建 kafka.javaapi.consumer.ZookeeperConsumerConnector
    consumer = Consumer.createJavaConsumerConnector(
      createConsumerConfig(
        zookeeper,
        groupId));
    this.topic = topic;
  }
  public void shutdown() {
    if (consumer != null)
      consumer.shutdown();
    if (executor != null)
      executor.shutdown();
  }
  public void run(int numThreads) {
    Map<String, Integer> topicCountMap = new HashMap<String, Integer>();
    //设置 Topic 和 Stream 的映射，其中 Key 为 Topic，Value 为 Stream 的个数
    topicCountMap.put(topic, new Integer(numThreads));
    //按照设置的映射关系创建若干个 KafkaStream
    Map<String, List<KafkaStream<byte[], byte[]>>> consumerMap =
      consumer.createMessageStreams(topicCountMap);
    List<KafkaStream<byte[], byte[]>> streams = consumerMap.get(topic);
    //设置 numThreads 个线程
    executor = Executors.newFixedThreadPool(numThreads);
    int threadNumber = 0;
    for (final KafkaStream stream : streams) {
      //开启消费者线程针对 stream 进行处理
      executor.submit(new ConsumerMsgTask(stream, threadNumber));
      threadNumber++;
    }
  }
  private static ConsumerConfig createConsumerConfig(String zookeeper, StringgroupId) {
    Properties props = new Properties();
    /* 只需要配置 Zookeeper 地址即可，因为高级消费者会主动从 Zookeeper 上获取 Kafka 集群的信息 */
    props.put("zookeeper.connect", zookeeper);
    //配置 Consumer Group
    props.put("group.id", groupId);
    //配置自动提交偏移量的时间间隔
    props.put("auto.commit.interval.ms", "1000");
    return new ConsumerConfig(props);
  }
  public static void main(String[] arg) {
```

```
        String[] args = {"172.23.8.160:2188", "Group-1", "Topic-0", "4"};
        String zooKeeper = args[0];
        String groupId = args[1];
        String topic = args[2];
        int threads = Integer.parseInt(args[3]);
        //初始化 KafkaHighConsumer
        KafkaHighConsumer demo = new KafkaHighConsumer(zooKeeper, groupId, topic);
        try {
          demo.run(threads);
          //客户端启动消费线程之后可以进行其他处理
        } catch (Exception e) {
          //根据业务的具体异常进行处理
          ;
        }
        //释放资源
        demo.shutdown();
      }
    }
```

当启动 ConsumerMsgTask 线程之后，接下来就是针对每个线程各自拥有的 KafkaStream 进行消息的处理：

```
    public class ConsumerMsgTask implements Runnable {
      private KafkaStream stream;
      private int threadNumber;
      public ConsumerMsgTask(KafkaStream stream, int threadNumber) {
        this.threadNumber = threadNumber;
        this.stream = stream;
      }
      public void run() {
        //获取 KafkaStream 的迭代器
        ConsumerIterator<byte[], byte[]> it = stream.iterator();
        //遍历迭代器获取消息
        while (it.hasNext())
          System.out.println("Thread " + threadNumber + ": "+ new String
            (it.next().message()));
        System.out.println("Shutting down Thread: " + threadNumber);
      }
    }
```

因此最终是通过遍历 KafkaStream 的迭代器来进行消息的读取。

8.2.4 原理解析

消息的最终获取是通过遍历 KafkaStream 的迭代器 ConsumerIterator 来逐条获取的，其数据来源于分配给该 KafkaStream 的阻塞消息队列 BlockingQueue，而 BlockingQueue 的数据来源于针对每个 Broker Server 的 FetchThread 线程。FetchThread 线程会将 Broker Server 上的部分 Partition 数据发送给对应的阻塞消息队列 BlockingQueue，而 KafkaStream 正

是从该阻塞消息队列 BlockingQueue 中不断地消费消息，其内部具体的实现原理如图 8-3 所示。

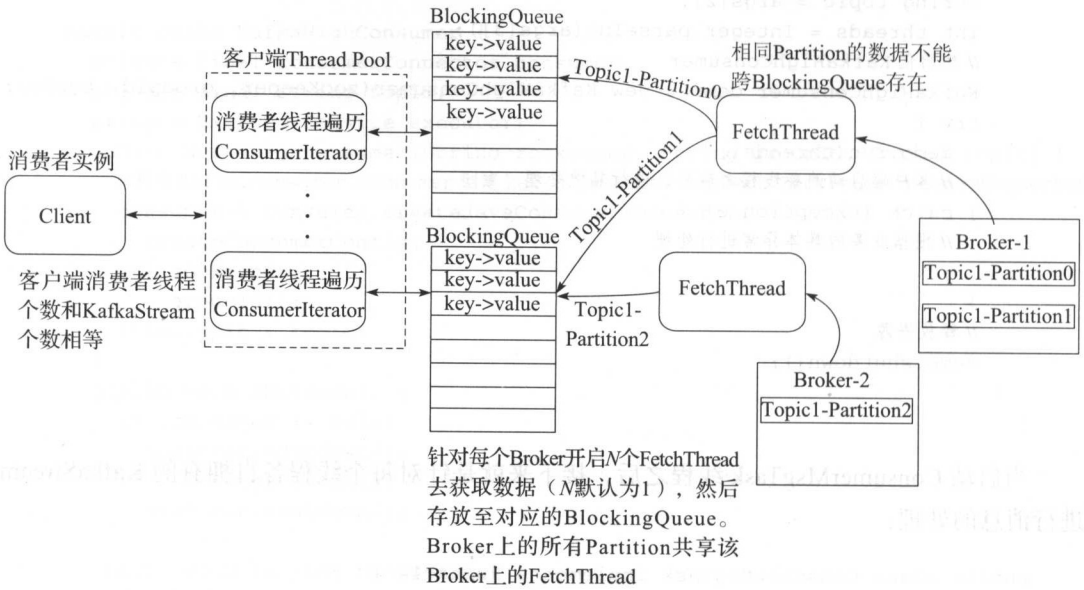


图 8-3 高级消费者内部实现原理

其中，kafka.javaapi.consumer.ZookeeperConsumerConnector 提供了所有功能，它仅仅是对 kafka.consumer.ZookeeperConsumerConnector 进行了一层包装，如下所示：

```
private[kafka] class ZookeeperConsumerConnector(
  val config: ConsumerConfig,
  val enableFetcher: Boolean)  extends ConsumerConnector {
  // 封装了 kafka.consumer.ZookeeperConsumerConnector
  private val underlying = new kafka.consumer.ZookeeperConsumerConnector(config,
    enableFetcher)
  private val messageStreamCreated = new AtomicBoolean(false)
  def createMessageStreams[K,V](
    topicCountMap: java.util.Map[String,java.lang.Integer],
    keyDecoder: Decoder[K],
    valueDecoder: Decoder[V]) : java.util.Map[String,java.util.
      List[KafkaStream[K,V]]] = {
      if (messageStreamCreated.getAndSet(true))
        throw new MessageStreamsExistException(this.getClass.getSimpleName +
          " can create message streams at most once",null)
      val scalaTopicCountMap: Map[String, Int] = {
        import JavaConversions._
        Map.empty[String, Int] ++ (
          topicCountMap.asInstanceOf[java.util.Map[String, Int]]: mutable.
            Map[String,Int])
```

```
        }
        // 利用 kafka.consumer.ZookeeperConsumerConnector 来创建真正的 KafkaStream
        val scalaReturn = underlying.consume(scalaTopicCountMap, keyDecoder, valueDecoder)
        val ret = new java.util.HashMap[String,java.util.List[KafkaStream[K,V]]]
        for ((topic, streams) <- scalaReturn) {
          var javaStreamList = new java.util.ArrayList[KafkaStream[K,V]]
        for (stream <- streams)
          javaStreamList.add(stream)
        ret.put(topic, javaStreamList)
      }
      ret
    }
    ......
}
```

可见真正实现高级消费者的类为 kafka.consumer.ZookeeperConsumerConnector，接下来将围绕它来讲解涉及图 8-3 中的四个方面：

1）ConsumerThread 和 Partition 的分配算法。

2）FetchThread 的启动过程。

3）KafkaStream 如何遍历 BlockingQueue。

4）KafkaStream 的负载均衡流程。

8.2.4.1 ConsumerThread 和 Partition 的分配算法

ConsumerThread 本质上就是客户端的消费者线程，每一个消费者线程消费若干个 Partition 上的数据或者没有消费数据，并且 ConsumerThread 和 BlockingQueue 相互之间一一映射，只要确定了 ConsumerThread 和 Partition 之间的关系，也就确定了 BlockingQueue 和 Partition 之间的关系。

Kafka 提供了两种 ConsumerThread 和 Partition 的分配算法，分别为 Range（范围分区分配）和 RoundRobin（循环分区分配），分配算法由参数 partition.assignment.strategy 决定，默认为 range。Range（范围分区分配）算法由 RangeAssignor 类实现，RoundRobin（循环分区分配）算法由 RoundRobinAssignor 类实现。

1. RoundRobin 分配算法

RoundRobin 分配算法有一个前提条件：同一个消费组下的每个消费者实例订阅的 Topic 都相同，且每个 Topic 对应的消费者线程个数也相同。主要是为了尽可能保证消费的时候负载均衡，不仅仅是每个消费者实例之间的负载均衡，而且也是每个消费者实例内部各个消费者线程之间的负载均衡。

假设当前存在某个 Topic，其拥有 6 个 Partition，建立 2 个消费者实例，这两个消费者实例同属于相同的消费者组 Group0，每个消费者实例分别有 2 个消费者线程订阅该 Topic，则其分配的流程如下所示。

1）统计 Topic 的分区：Partition-0、Partition-1、Partition-2、Partition-3、Partition-4、

Partition-5。

2）统计消费者线程：Group0- 消费者实例 0- 消费者线程 0，Group0- 消费者实例 0- 消费者线程 1、Group0- 消费者实例 1- 消费者线程 0，Group0- 消费者实例 1- 消费者线程 1。

3）将 Topic 的分区按照 Hash 值排序，排序结果如下：Partition-1，Partition-3，Partition-2，Partition-4，Partition-0，Partition-5。

4）将排序之后的 Topic 分区循环分配给消费者线程，分配结果如下：Group0- 消费者实例 0- 消费者线程 0 负责 Partition-1 和 Partition-0，Group0- 消费者实例 0- 消费者线程 1 负责 Partition-3 和 Partition-5，Group0- 消费者实例 1- 消费者线程 0 负责 Partition-2，Group0- 消费者实例 1- 消费者线程 1 负责 Partition-4。

在上述分配流程中，目标 Topic 的分区按照 Topic+Partition 取 Hash 值升序排序，然后将排序之后的 Partition 列表按照循环遍历的方式分配到具体的消费者线程。其具体实现如下所示：

```
class RoundRobinAssignor() extends PartitionAssignor with Logging {
  def assign(ctx: AssignmentContext) = {
    val partitionOwnershipDecision = collection.mutable.Map[TopicAndPartition,
      ConsumerThreadId]()
    /*
      AssignmentContext 内部的 consumersForTopic 来自 Zookeeper 目录 /consumers/
      [group]/ids/;
      里面记录了相同 group 下的不同消费者实例订阅的 Topic 详情;
      遍历 /consumers/[group]/ids/ 下每一个具体的 id，收集相同 group 下的 Topic 订阅详情
     */
     val (headTopic, headThreadIdSet) = (
      ctx.consumersForTopic.head._1,
      ctx.consumersForTopic.head._2.toSet)
    ctx.consumersForTopic.foreach { case (topic, threadIds) =>
      val threadIdSet = threadIds.toSet
      /*
        相同 group 下的不同 Topic 的 ConsumerThreadId 集合必须一样：
        1）每个消费者实例订阅的 Topic 必须一样
        2）Topic 对应的消费者线程分布也相同
       */
       require(
        threadIdSet == headThreadIdSet,
        "Round-robin assignment is allowed only if all consumers in the group
          subscribe to the same topics, " +
          "AND if the stream counts across topics are identical for a given consumer
            instance.\n" +
          "Topic %s has the following available consumer streams: %s\n".format(topic,
            threadIdSet) +
          "Topic %s has the following available consumer streams: %s\n".
            format(headTopic, headThreadIdSet))
    }
    // 将消费者线程集合排序，然后创建一个轮询的迭代器
```

```
    val threadAssignor = Utils.circularIterator(headThreadIdSet.toSeq.sorted)
    //将 Partition 升序排序
    val allTopicPartitions = ctx.partitionsForTopic.flatMap {
      case(topic, partitions) =>
        info("Consumer %s rebalancing the following partitions for topic %s: %s"
          .format(ctx.consumerId, topic, partitions))
        partitions.map(
          partition => {TopicAndPartition(topic, partition)})
    }.toSeq.sortWith((topicPartition1, topicPartition2) => {
    //利用 Hash 值排序是为了降低 Topic 的所有 Partition 落在相同的消费者实例上的可能性
      topicPartition1.toString.hashCode < topicPartition2.toString.hashCode
    })
    //遍历该 partition 列表
    allTopicPartitions.foreach(topicPartition => {
      //迭代消费者线程集合
      val threadId = threadAssignor.next()
      //如果属于当前的消费者实例，则记录其 topicPartition 和消费者线程的关系
      if (threadId.consumer == ctx.consumerId)
        partitionOwnershipDecision += (topicPartition -> threadId)
    })
    partitionOwnershipDecision
  }
}
```

2. Range 分配算法

Range 分配算法就是将一定范围之内的 Partition 分配给单个消费者线程，也就是说消费者线程会消费前后相邻的 Partition。

假设当前存在某个 Topic，其拥有 6 个 Partition，建立 2 个消费者实例，这两个消费者实例同属于相同的消费者组 Group0，每个消费者实例分别有 2 个消费者线程订阅该 Topic，则其分配流程如下所示：

1）统计 Topic 的分区：Partition-0、Partition-1、Partition-2、Partition-3、Partition-4、Partition-5。

2）统计消费者线程：Group0- 消费者实例 0- 消费者线程 0，Group0- 消费者实例 0- 消费者线程 1，Group0- 消费者实例 1- 消费者线程 0，Group0- 消费者实例 1- 消费者线程 1。

3）将 Topic 的分区按范围分配给消费者线程，分配结果如下：Group0- 消费者实例 0- 消费者线程 0 负责 Partition-0 和 Partition-1，Group0- 消费者实例 0- 消费者线程 1 负责 Partition-2 和 Partition-3，Group0- 消费者实例 1- 消费者线程 0 负责 Partition-4，Group0- 消费者实例 1- 消费者线程 1 负责 Partition-5。

在上述分配流程中，前后相邻的 Partition 可能会分配给同一个消费者线程，其具体实现如下所示：

```
class RangeAssignor() extends PartitionAssignor with Logging {
  def assign(ctx: AssignmentContext) = {
    val partitionOwnershipDecision = collection.mutable.Map[TopicAndPartition,
```

```
        ConsumerThreadId]()
      //遍历当前消费者实例订阅每个 Topic 的消费者线程
      for ((topic, consumerThreadIdSet) <- ctx.myTopicThreadIds) {
        /*
         AssignmentContext 内部的 consumersForTopic 来自 Zookeeper 目录 /consumers/[group]/ids/
         里面记录了相同 group 下的不同消费者实例订阅的 Topic 详情;
         遍历 /consumers/[group]/ids/ 下每一个具体的 id，收集相同 group 下的 Topic 订阅详情
         然后取出某个 Topic 所有的消费者线程
        */
        val curConsumers = ctx.consumersForTopic(topic)
        //从 Zookeeper 目录的 /brokers/topics/[topic] 加载该 Topic 的所有分区
        val curPartitions: Seq[Int] = ctx.partitionsForTopic(topic)
        //计算出平均每个消费者线程消费几个分区
        val nPartsPerConsumer = curPartitions.size / curConsumers.size
        //计算出还需几个消费者线程消费剩余没有平均分配到的分区
        val nConsumersWithExtraPart = curPartitions.size % curConsumers.size
        //遍历特定 Topic 的消费者线程
        for (consumerThreadId <- consumerThreadIdSet) {
          //取出其位置索引
          val myConsumerPosition = curConsumers.indexOf(consumerThreadId)
          assert(myConsumerPosition >= 0)
          //计算起始分区索引
          val startPart = nPartsPerConsumer * myConsumerPosition + myConsumerPosition.
           min(nConsumersWithExtraPart)
          //计算分区个数
          val nParts = nPartsPerConsumer +
            (if (myConsumerPosition + 1 > nConsumersWithExtraPart) 0 else 1)
          //如果消费分区个数为 0，则不消费，否则消费
          if (nParts <= 0)
            warn("No broker partitions consumed by consumer thread " + consumerThreadId
              + " for topic " + topic)
          else {
            //记录 topicPartition 和消费者线程的关系
            for (i <- startPart until startPart + nParts) {
              val partition = curPartitions(i)
              partitionOwnershipDecision += (TopicAndPartition(topic, partition) ->
                consumerThreadId)
            }
          }
        }
      }
      partitionOwnershipDecision
    }
  }
```

8.2.4.2 ConsumerFetcherThread 的消费流程

一旦当前消费者实例的 ConsumerThread 和 Partition 的关系决定之后，就需要启动 ConsumerFetcherThread 消费 Broker Server 上的 Partition 的消息。每个 Broker Server 对应 num.consumer.fetchers 个 ConsumerFetcherThread，默认为 1。ConsumerFetcherThread 会将

获取到的 Partition 数据转发至对应的 BlockingQueue 供 ConsumerThread 获取消息。

回顾 8.1.2 节简单消费者流程中客户端需要知道 Partition 的 Leader Replica 所在的 Broker Server，因此需要主动发送 TopicMetadataRequest 询问 Kafka 集群相关 Topic 的元数据。而在高级消费者流程中，这部分工作由内部一个称为 LeaderFinderThread 的线程完成，该线程负责寻找 Partition 的 Leader Replica 所在的 Broker Server，一旦找到之后，则会向对应的 ConsumerFetcherThread 下发拉取该 Partition 消息的指令。

因此在本节中将分 2 个部分介绍：1）ConsumerFetcherThread 的启动；2）ConsumerFetcherThread 的执行逻辑。这两部分都是由 ConsumerFetcherManager 模块负责的。

1. ConsumerFetcherThread 的启动

ConsumerFetcherManager 在启动的时候会建立 LeaderFinderThread，其中 ConsumerFetcherManager 内部的 noLeaderPartitionSet 变量保存了当前其 Leader Replica 还未明确的 TopicAndPartition，LeaderFinderThread 从 noLeaderPartitionSet 获取具体的 TopicAndPartition，然后遍历 Kafka 集群中的 Broker Server，向其发送 TopicMetadataRequest 来获取相关 Topic 的元数据信息。其具体实现如下：

```
private class LeaderFinderThread(name: String) extends ShutdownableThread(name) {
  override def doWork() {
    // 保存 TopicAndPartition 的 Leader Replica 所在的 Broker Server
    val leaderForPartitionsMap = new HashMap[TopicAndPartition, Broker]
    lock.lock()
    try {
      // 如果当前没有待寻找的，则等待
      while (noLeaderPartitionSet.isEmpty) {
        cond.await()
      }
      // 从 Zookeeper 的 /brokers/ids 目录获取当前 Kafka 集群中所有的 Broker Server
      val brokers = getAllBrokersInCluster(zkClient)
      /* 遍历 Broker Server 列表，查找 noLeaderPartitionSet 中的 topic 元数据，如果最终没
         有找到，则抛出异常 */
        val topicsMetadata = ClientUtils.fetchTopicMetadata(
        noLeaderPartitionSet.map(m => m.topic).toSet,
        brokers,
        config.clientId,
        config.socketTimeoutMs,
        correlationId.getAndIncrement).topicsMetadata
      topicsMetadata.foreach { tmd =>
        val topic = tmd.topic
        tmd.partitionsMetadata.foreach { pmd =>
          val topicAndPartition = TopicAndPartition(topic, pmd.partitionId)
          if(pmd.leader.isDefined && noLeaderPartitionSet.
            contains(topicAndPartition)) {
            val leaderBroker = pmd.leader.get
            // 更新 leaderForPartitionsMap
            leaderForPartitionsMap.put(topicAndPartition, leaderBroker)
```

```
                //将 topicAndPartition 从 noLeaderPartitionSet 中剔除
                noLeaderPartitionSet -= topicAndPartition
              }
            }
          }
        } catch {
          case t: Throwable => {
            if (!isRunning.get())
              throw t
            else
              warn("Failed to find leader for %s".format(noLeaderPartitionSet), t)
            }
          } finally {
            lock.unlock()
          }
        try {
          /* 将 topicAndPartition 添加至对应的 ConsumerFetcherThread，如果 ConsumerFetcherThread
            不存在，则创建 */
          addFetcherForPartitions(leaderForPartitionsMap.map{
            case (topicAndPartition, broker) =>
              topicAndPartition -> BrokerAndInitialOffset(
                broker,
                partitionMap(topicAndPartition).getFetchOffset())}
            )
          } catch {
            case t: Throwable => {
              if (!isRunning.get())
                throw t
              else {
                lock.lock()
                //addFetcherForPartitions 失败，则需要继续寻找 Leader Replica
                noLeaderPartitionSet ++= leaderForPartitionsMap.keySet
                lock.unlock()
              }
            }
          }
          //关闭空闲的 fetcher 线程
          shutdownIdleFetcherThreads()
          Thread.sleep(config.refreshLeaderBackoffMs)
        }
      }
    }
```

其中 addFetcherForPartitions 会负责启动 ConsumerFetcherThread，如果已经启动的话，则会利用 topicAndPartition 和对应的 offset 更新 ConsumerFetcherThread 内部的 partitionMap，该 partition Map 里面保存了该 ConsumerFetcherThread 需要消费的 topicAndPartition 和起始偏移量。addFetcherForPartitions 内部实现如下：

```
def addFetcherForPartitions(
```

```
partitionAndOffsets: Map[TopicAndPartition,
BrokerAndInitialOffset]) {
mapLock synchronized {
  /*
    将 partitionAndOffsets 按照 BrokerAndFetcherId 分组，其中 BrokerAndFetcherId 中的
    broker 为 Leader Replica 所在的 Broker;
    fetcherId 为 Utils.abs(31 * topic.hashCode() + partitionId) % numFetchers，其中
    numFetchers=num.consumer.fetchers 目的为计算出该 topicAndPartition 由具体哪个
    ConsumerFetcherThread 负责
   */
  val partitionsPerFetcher = partitionAndOffsets.groupBy{
    case(topicAndPartition, brokerAndInitialOffset) =>
      BrokerAndFetcherId(
        brokerAndInitialOffset.broker,
        getFetcherId(topicAndPartition.topic, topicAndPartition.partition))}
  for ((brokerAndFetcherId, partitionAndOffsets) <- partitionsPerFetcher) {
    var fetcherThread: AbstractFetcherThread = null
    // fetcherThreadMap 保存了 BrokerAndFetcherId 和 ConsumerFetcherThread 的映射关系
    fetcherThreadMap.get(brokerAndFetcherId) match {
      // ConsumerFetcherThread 已经存在，则不做任何事情
      case Some(f) => fetcherThread = f
      /*ConsumerFetcherThread 不存在，创建 ConsumerFetcherThread 并启动，同时更新
        fetcherThreadMap*/
      case None =>
        fetcherThread = createFetcherThread(
          brokerAndFetcherId.fetcherId,
          brokerAndFetcherId.broker)
        fetcherThreadMap.put(brokerAndFetcherId, fetcherThread)
        fetcherThread.start
      }
    // 更新 ConsumerFetcherThread 内部的 partitionMap
    fetcherThreadMap(brokerAndFetcherId).addPartitions(
      partitionAndOffsets.map {
        case (topicAndPartition, brokerAndInitOffset) =>
          topicAndPartition -> brokerAndInitOffset.initOffset})
  }
 }
}
```

2. ConsumerFetcherThread 的执行逻辑

ConsumerFetcherThread 会遍历其内部的 partitionMap，消费 partitionMap 中包含的 topicAndPartition，然后将 topicAndPartition 的数据转发至相应的 BlockingQueue。

ConsumerFetcherThread 继承自 AbstractFetcherThread，AbstractFetcherThread 内部的 doWork 流程负责提取 partitionMap 中的 topicAndPartition 和 offset，向 Broker Server 发送 FetchRequest 请求，然后更新 partitionMap 中的 offset，最后利用 ConsumerFetcherThread 的 processPartitionData 函数来处理获取到的分区数据。

AbstractFetcherThread 的 doWork 的具体实现如下：

```
override def doWork() {
  inLock(partitionMapLock) {
    //partitionMap 为空，则等待
    if (partitionMap.isEmpty)
      partitionMapCond.await(200L, TimeUnit.MILLISECONDS)
    //遍历 partitionMap，组装 FetchRequest 请求参数
    partitionMap.foreach {
      case((topicAndPartition, offset)) =>
        fetchRequestBuilder.addFetch(
          topicAndPartition.topic,
          topicAndPartition.partition,
          offset, fetchSize)
    }
  }
  val fetchRequest = fetchRequestBuilder.build()
  if (!fetchRequest.requestInfo.isEmpty)
    //处理 FetchRequest 请求
    processFetchRequest(fetchRequest)
}
private def processFetchRequest(fetchRequest: FetchRequest) {
  val partitionsWithError = new mutable.HashSet[TopicAndPartition]
  var response: FetchResponse = null
  try {
    /* 获取消息时本质上还是利用普通消费者 API 实现的，即向 Broker Server 发送 FetchRequest 请求 */
      response = simpleConsumer.fetch(fetchRequest)
  } catch {
    case t: Throwable =>
      if (isRunning.get) {
        partitionMapLock synchronized {
          //发送异常，记录 partitionsWithError
          partitionsWithError ++= partitionMap.keys
        }
      }
  }
  //记录发送频率
  fetcherStats.requestRate.mark()
  if (response != null) {
    inLock(partitionMapLock) {
      //遍历 FetchResponse
      response.data.foreach {
        case(topicAndPartition, partitionData) =>
          val (topic, partitionId) = topicAndPartition.asTuple
          val currentOffset = partitionMap.get(topicAndPartition)
          /* 校验 partitionMap 中的偏移量和 FetchRequest 中的偏移量是否一致，一致则表明
             响应有效 */
          if (currentOffset.isDefined && fetchRequest.
            requestInfo(topicAndPartition).offset
            == currentOffset.get) {
```

```
        partitionData.error match {
          // 响应成功
          case ErrorMapping.NoError =>
            try {
              // 获取该 topicAndPartition 的 ByteBufferMessageSet
              val messages = partitionData.messages.asInstanceOf[ByteBuffe
                rMessageSet]
              // 获取 ByteBufferMessageSet 的有效字节数
              val validBytes = messages.validBytes
             // 获取 ByteBufferMessageSet 的下一个偏移量
              val newOffset = messages.shallowIterator.toSeq.lastOption match {
                case Some(m: MessageAndOffset) => m.nextOffset
                case None => currentOffset.get
              }
              // 更新 partitionMap
              partitionMap.put(topicAndPartition, newOffset)
              ......
              /* 调用 ConsumerFetcherThread 的 processPartitionData 流程，本质上就是
                 将 partitionData 转发至 BlockingQueue*/
              processPartitionData(topicAndPartition, currentOffset.get,
                partitionData)
            } catch {
              // 忽略 InvalidMessageException 异常
              case ime: InvalidMessageException => ……
              // processPartitionData 中抛出异常，则继续往上抛出
              case e: Throwable =>
                throw new KafkaException("error processing data for partition
                [%s,%d] offset %d".format(topic, partitionId,
                currentOffset.get), e)
            }
          case ErrorMapping.OffsetOutOfRangeCode =>
            try {
              /* 偏移量越界，重置偏移量，根据 auto.offset.reset 策略选择不同的重置策略，
                 分别为 EarliestTime 和 LatestTime*/
              val newOffset = handleOffsetOutOfRange(topicAndPartition)
              partitionMap.put(topicAndPartition, newOffset)
            } catch {
              case e: Throwable =>
                // 重置失败，则添加入 partitionsWithError
                partitionsWithError += topicAndPartition
            }
          case _ =>
            /* 其他异常，ConsumerFetcherThread 正在运行的情况下也更新
              partitionsWithError*/
            if (isRunning.get) {
              partitionsWithError += topicAndPartition
            }
        }
      }
    }
```

```
    }
  }
  if(partitionsWithError.size > 0) {
    /* 根据 partitionsWithError，将 topicAndPartition 从 partitionMap 删除，并添加入
       noLeaderPartitionSet，重新寻找 Leader replica*/
    handlePartitionsWithErrors(partitionsWithError)
  }
}
```

ConsumerFetcherThread 中的 partitionMap 参数保存了 TopicAndPartition 和 PartitionTopicInfo 的映射关系，其中 PartitionTopicInfo 中的 chunkQueue 参数指定了该 TopicAndPartition 所对应的 BlockingQueue。processPartitionData 负责从 PartitionTopicInfo 中获取 chunkQueue，然后将 ByteBufferMessageSet 消息集合存放入 chunkQueue，其实现过程如下：

```
def processPartitionData(
  topicAndPartition: TopicAndPartition,
  fetchOffset: Long,
  partitionData: FetchResponsePartitionData) {
  // 获取 PartitionTopicInfo
  val pti = partitionMap(topicAndPartition)
  // 校验 fetchOffset 前后是否一致
  if (pti.getFetchOffset != fetchOffset)
    throw new RuntimeException("Offset doesn't match for partition [%s,%d] pti
      offset: %d fetch offset: %d".format(topicAndPartition.topic,
      topicAndPartition.partition, pti.getFetchOffset, fetchOffset))
  /*
    调用 PartitionTopicInfo 的 enqueue 流程：
    1）将 ByteBufferMessageSet 存放入 BlockingQueue
    2）更新 PartitionTopicInfo 中的 fetchedOffset
   */
  pti.enqueue(partitionData.messages.asInstanceOf[ByteBufferMessageSet])
}
```

可见 BlockingQueue 中的每一个元素都是一定偏移量范围内的消息集合。

8.2.4.3 KafkaStream 遍历 Blocking Queue 的流程

在上个小节中提到：BlockingQueue 中的每一个元素都是一定偏移量范围内的消息集合，即 FetchedDataChunk。FatchedDataChunk 是消息数据块，里面包含了一定数量的消息条目组成如下所示：

```
case class FetchedDataChunk(messages: ByteBufferMessageSet,
                            topicInfo: PartitionTopicInfo,
                            fetchOffset: Long)
```

其中 messages 利用 byte buffer 存储了某个偏移量范围之内的所有消息。

高级消费者消费消息的时候是利用 KafkaStream 的迭代器来实现的，即 ConsumerIterator。该迭代器提供了遍历 BlockingQueue 中的每一条消息的能力。考虑到 BlockingQueue 的组成

和 FetchedDataChunk 的组成，则 ConsumerIterator 消费某条消息的流程如图 8-4 所示。

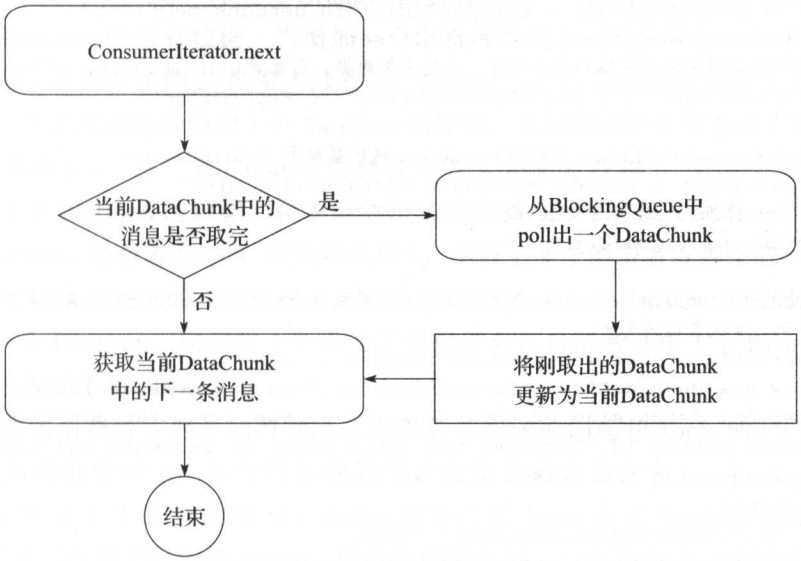


图 8-4 ConsumerIterator 的实现原理

以上 ConsumerIterator 的实现原理对应的正是 ConsumerIterator 内部的 makeNext 流程，这个流程提供了连续访问下一条消息的能力，其具体实现如下：

```
protected def makeNext(): MessageAndMetadata[K, V] = {
  var currentDataChunk: FetchedDataChunk = null
  // current 为 MessageAndOffset 本身的迭代器
  var localCurrent = current.get()
  /* 如果 current 为 null，或者 MessageAndOffset 本身已经遍历完了，则需要重新从
     BlockingQueue 中拉取 MessageAndOffset*/
  if(localCurrent == null || !localCurrent.hasNext) {
    // consumer.timeout.ms 参数 <0，则阻塞获取 MessageAndOffset
    if (consumerTimeoutMs < 0)
      currentDataChunk = channel.take
    else {// consumer.timeout.ms 参数 >0，则获取 MessageAndOffset，如果时间超时则返回
      currentDataChunk = channel.poll(consumerTimeoutMs, TimeUnit.MILLISECONDS)
      if (currentDataChunk == null) {
        // 没有获取到 MessageAndOffset，则置迭代器状态为 NOT_READY
        resetState()
        // 向上抛出消费超时的异常
        throw new ConsumerTimeoutException
      }
    }
    // 接收到关闭的指令，则置迭代器状态为 DONE，立刻返回
    if(currentDataChunk eq ZookeeperConsumerConnector.shutdownCommand) {
      return allDone
    } else {// 获取到一个 MessageAndOffset
```

```
    currentTopicInfo = currentDataChunk.topicInfo
    val cdcFetchOffset = currentDataChunk.fetchOffset
    val ctiConsumeOffset = currentTopicInfo.getConsumeOffset
    if (ctiConsumeOffset < cdcFetchOffset) {
      //预期偏移量和实际偏移量不相等，可能丢失数据，则重置 ConsumeOffset 为 FetchOffset
      currentTopicInfo.resetConsumeOffset(cdcFetchOffset)
    }
    //将 MessageAndOffset 内部的 messages 迭代器赋予 localCurrent
    localCurrent = currentDataChunk.messages.iterator
    //将 localCurrent 赋予 current
    current.set(localCurrent)
  }
  /*fetch.message.max.bytes 参数设置过小，导致 currentDataChunk 为消息字节数为 0，没有
    获取到一条完整的消息 */
  if(currentDataChunk.messages.validBytes == 0)
    throw new MessageSizeTooLargeException("Found a message larger than the
      maximum fetch size of this consumer on topic " + "%s partition %d at
      fetch offset %d. Increase the fetch size, or decrease the maximum
      message size the broker will allow."
      .format(
        currentDataChunk.topicInfo.topic,
        currentDataChunk.topicInfo.partitionId,
        currentDataChunk.fetchOffset))
  }
  //获取 localCurrent 中的下一条消息
  var item = localCurrent.next()
  //过滤已经消费过的消息
  while (item.offset < currentTopicInfo.getConsumeOffset && localCurrent.hasNext) {
    item = localCurrent.next()
  }
  //获取下一条消息
  consumedOffset = item.nextOffset
  item.message.ensureValid()
  //组装返回格式
  new MessageAndMetadata(
    currentTopicInfo.topic,
    currentTopicInfo.partitionId,
    item.message,
    item.offset,
    keyDecoder,
    valueDecoder)
}
```

可见 ConsumerIterator 通过先遍历当前的 ByteBufferMessageSet 内部的消息，如果已经遍历完，则重置当前的 ByteBufferMessageSet 为 BlockingQueue 中的下一个 ByteBufferMessageSet，然后继续遍历来提供客户端逐条访问消息的能力。

8.2.4.4 KafkaStream 的负载均衡流程

KafkaStream 的负载均衡流程指的是相同消费组下的不同消费者线程可以动态消费

Topic 的数据。假设 topic1 有 0、1、2 共三个 Partition，当 Group1 只有一个消费者实例 1，该实例有一个消费者线程 1，称之为 Group1- 消费者实例 1- 消费者线程 1，该消费者线程可消费这 3 个 Partition 的所有数据。当增加一个消费者实例 2，该实例有一个消费者线程 1，称之为 Group1- 消费者实例 2- 消费者线程 1，则其中一个消费者线程消费 2 个 Partition 的数据，另外一个消费者线程消费 1 个 Partition 的数据。此时继续增加消费者实例 3，该实例有一个消费者线程 1，称之为 Group1- 消费者实例 3- 消费者线程 1，则 3 个消费者线程分别消费其中 1 个 Partition 的数据。接着继续增加消费者实例 4，该实例有一个消费者线程 1，称之为 Group1- 消费者实例 4- 消费者线程 1，则以上 4 个消费者线程中有一个消费者线程不消费任何数据，剩下的 3 个消费者线程各自消费其中 1 个 Partition 的数据。然后针对 topic1 增加一个 Partition，则此时 4 个消费者线程将会各自消费其中 1 个 Partition 的数据。以上的变化过程如图 8-5 所示。

那么具体的消费者实例是如何知道其他消费者实例的创建和 Topic 的 Partition 个数的变化呢？在高级消费者中，每个具体的消费者实例启动之后会在 /consumers/[group]/ids/ 的 Zookeeper 目录下注册自己的 id；Kafka 集群内部 Topic 会在 /brokers/topics/[topic]/ 的 Zookeeper 目录下注册自己的 Partition，因此消费者实例一旦发现以上 2 个路径的数据发生变化时，则会触发高级消费者的负载均衡流程，除此之外，消费者实例一旦和 Zookeeper 的链接重新建立时也会触发高级消费者的负载均衡流程。但是此种方式存在以下几个弊端：

- Partition 的变化或者 Consumer 的增减都会触发全部 Consumer 实例的 Rebalance。
- 每个 Consumer 分别单独通过 Zookeeper 判断哪些 Broker 和 Consumer 宕机了，那么不同 Consumer 在同一时刻从 Zookeeper 看到的 View 就可能不一样，这是由 Zookeeper 的特性决定的，这就会造成不正确的 Reblance 尝试。
- 所有的 Consumer 都并不知道其他 Consumer 的 Rebalance 是否成功，这可能会导致 Consumer 工作在一个不一致的状态。

以上问题都会在将来的版本修复。

当消费者实例发生负载均衡时，其主要步骤如下：

1）关闭所有的 ConsumerFetchThread 线程。

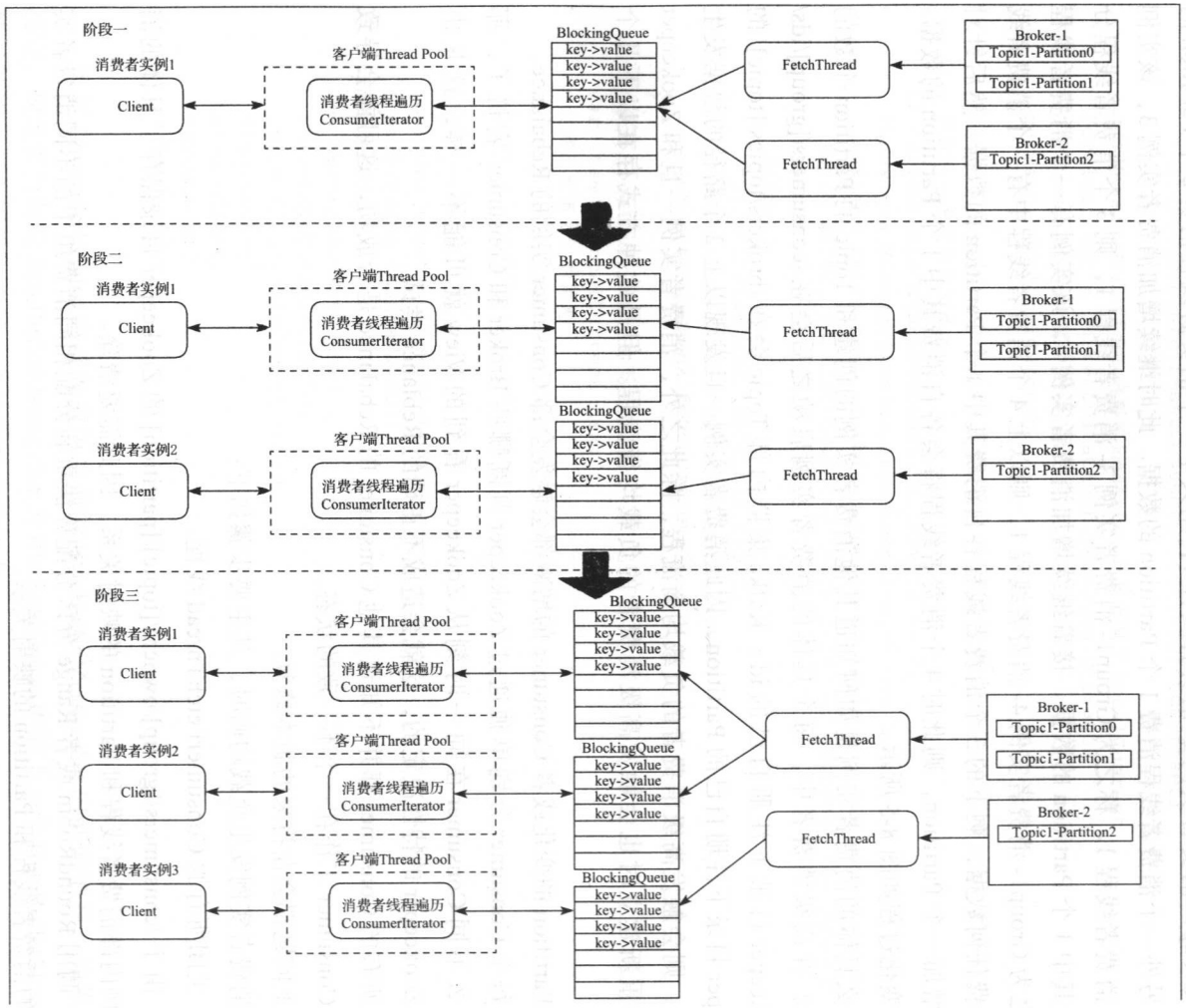
2）由于 /consumers/[group]/owners/[topic]/[partition] 的 Zookeeper 目录保存了具体的消费者实例内部消费者线程和 Partition 的映射关系，因此需要清除。

3）调用 RoundRobin 或者 Range 分区分配算法重新分配相同消费组下的不同消费者实例内部的消费者线程和 Partition 的映射关系。

4）更新每个 Partition 的起始偏移量。

5）更新 /consumers/[group]/owners/[topic]/[partition] 下 Partition 和消费者线程的关系。

6）重新启动所有的 ConsumerFetchThread 线程。


阶段一
消费者实例1
Client
客户端Thread Pool
消费者线程遍历
ConsumerIterator
BlockingQueue
key->value
key->value
key->value
key->value
FetchThread
FetchThread
Broker-1
Topic1-Partition0
Topic1-Partition1
Broker-2
Topic1-Partition2
阶段二
消费者实例1
Client
客户端Thread Pool
消费者线程遍历
ConsumerIterator
BlockingQueue
key->value
key->value
key->value
key->value
FetchThread
Broker-1
Topic1-Partition0
Topic1-Partition1
消费者实例2
Client
客户端Thread Pool
消费者线程遍历
ConsumerIterator
BlockingQueue
key->value
key->value
key->value
key->value
FetchThread
Broker-2
Topic1-Partition2
阶段三
消费者实例1
Client
客户端Thread Pool
消费者线程遍历
ConsumerIterator
BlockingQueue
key->value
key->value
key->value
key->value
FetchThread
Broker-1
Topic1-Partition0
Topic1-Partition1
消费者实例2
Client
客户端Thread Pool
消费者线程遍历
ConsumerIterator
BlockingQueue
key->value
key->value
key->value
key->value
FetchThread
Broker-2
Topic1-Partition2
消费者实例3
Client
客户端Thread Pool
消费者线程遍历
ConsumerIterator
BlockingQueue
key->value
key->value
key->value
key->value

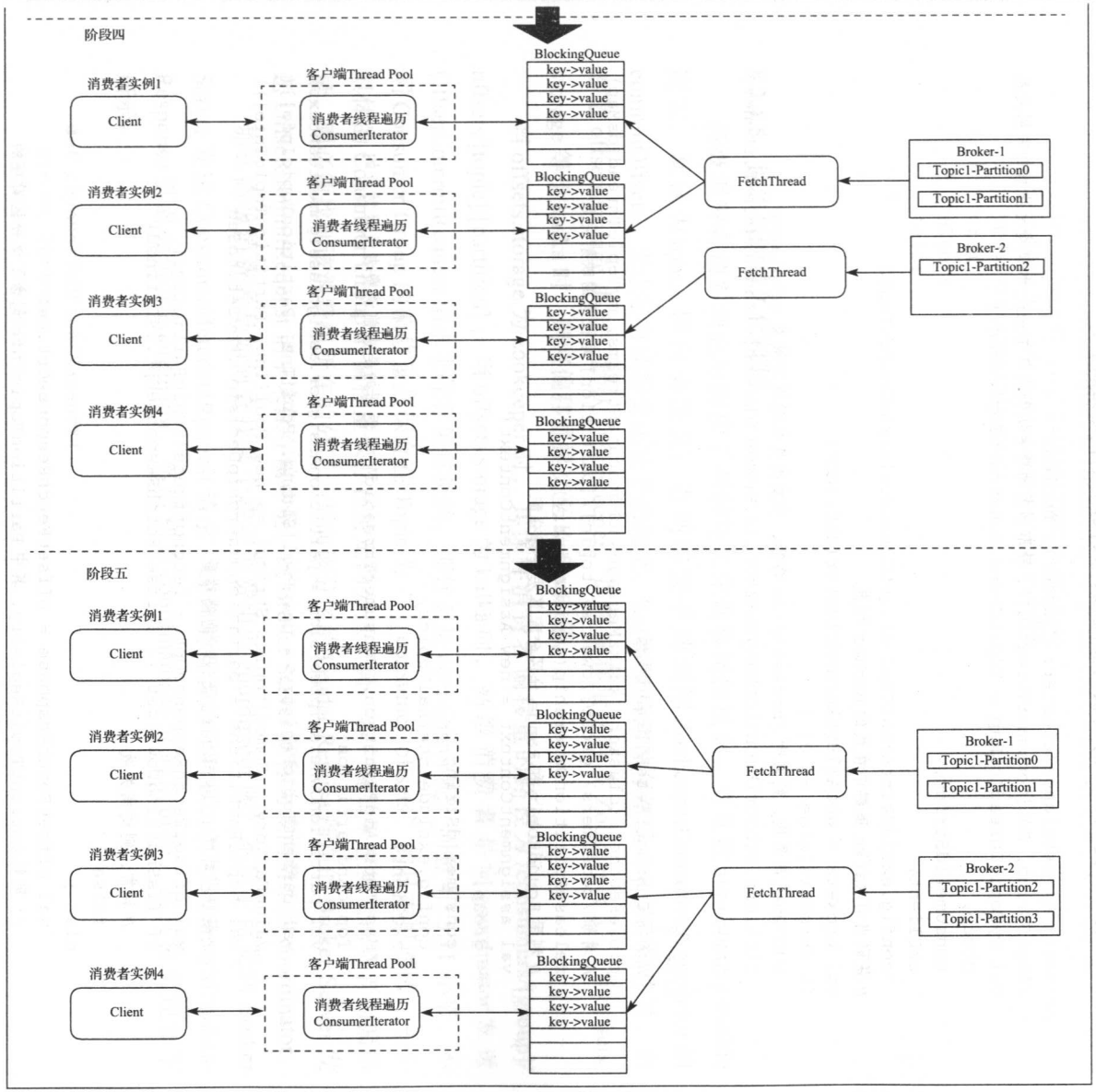


图 8-5　高级消费者的负载均衡流程

在高级消费者内部针对 Zookeeper 的连接建立、Topic 的 Partition 变化、Consumer 的新增，会分别建立 3 个不同的 Listener，分别为 ZKSessionExpireListener、ZKTopicPartitionChangeListener 和 ZKRebalancerListener，不过最终会通过 ZKRebalancerListener 的 rebalance 函数对外提供具体的负载均衡流程，其代码具体实现如下：

```
private def rebalance(cluster: Cluster): Boolean = {
  //Map[String, Set[ConsumerThreadId]]，获取当前消费者实例内部 Topic 和消费者线程的映射关系
  val myTopicThreadIdsMap = TopicCount.constructTopicCount(
    group,
    consumerIdString,
    zkClient,
    config.excludeInternalTopics).getConsumerThreadIdsPerTopic
  //获取当前 Kafka 集群中在线的 Broker 列表
  val brokers = getAllBrokersInCluster(zkClient)
  if (brokers.size == 0) {
    //Broker 都离线，则监听 /brokers/ids 路径，当发生变化时再次触发 rebalance
    zkClient.subscribeChildChanges(ZkUtils.BrokerIdsPath, loadBalancerListener)
    true
  }
  else {
    //关闭 ConsumerFetchThread 线程
    closeFetchers(cluster, kafkaMessageAndMetadataStreams, myTopicThreadIdsMap)
    //清除 /consumers/[group]/owners/[topic]/[partition] 路径下的映射关系
    releasePartitionOwnership(topicRegistry)
    //利用 RoundRobin 和 Range 分区分配算法进行分配
    val assignmentContext = new AssignmentContext(
      group,
      consumerIdString,
      config.excludeInternalTopics,
      zkClient)
    /*Map[TopicAndPartition, ConsumerThreadId] ，返回的是当前消费者线程对应的
      TopicAndPartition*/
      val partitionOwnershipDecision = partitionAssignor.assign(assignmentContext)
      val currentTopicRegistry = new Pool[String, Pool[Int, PartitionTopicInfo]](
      valueFactory = Some((topic: String) => new Pool[Int, PartitionTopicInfo]))
      val topicPartitions = partitionOwnershipDecision.keySet.toSeq
      //获取当前 TopicAndPartition 的偏移量
      val offsetFetchResponseOpt = fetchOffsets(topicPartitions)
      if (isShuttingDown.get || !offsetFetchResponseOpt.isDefined)
        //异常，则均衡失败
        false
      else {
        val offsetFetchResponse = offsetFetchResponseOpt.get
        /* 更新 currentTopicRegistry，其中 PartitionTopicInfo 包含了该分区对应的
           BlockingQueue 等信息 */
         topicPartitions.foreach(topicAndPartition => {
          val (topic, partition) = topicAndPartition.asTuple
          val offset = offsetFetchResponse.requestInfo(topicAndPartition).offset
          val threadId = partitionOwnershipDecision(topicAndPartition)
```

```
            addPartitionTopicInfo(currentTopicRegistry, partition, topic,
              offset, threadId)
          })
          // 更新 /consumers/[group]/owners/[topic]/[partition] 路径下的映射关系
          if(reflectPartitionOwnershipDecision(partitionOwnershipDecision)) {
            allTopicsOwnedPartitionsCount = partitionOwnershipDecision.size
            topicRegistry = currentTopicRegistry
            // 开始工作，启动 ConsumerFetchThread 线程
            updateFetcher(cluster)
            true
          } else {
            false
          }
        }
      }
    }
```

8.2.4.5 偏移量的持久化机制

高级消费者消费消息时提供了两种持久化偏移量的机制，由参数 auto.commit.enable 决定，默认为 true，即自动提交；否则需要手动调用 ZookeeperConsumerConnector 的 commitOffsets。无论是自动提交还是手动提交，Kafka 支持两种存储模式来保存偏移量，由参数 offsets.storage 决定，默认为 zookeeper，即将偏移量保存在 Zookeeper 上，当 offsets.storage 等于 kafka 时，则将偏移量提交至 Kafka 内部。

当 offsets.storage 为 zookeeper 时，其保存消费者偏移量的路径为 /consumers/[group]/offsets/[topic]/[partition]；当 offsets.storage 为 kafka 时，高级消费者是向 coordinator 发送 OffsetCommitRequest 请求来保存偏移量的。那什么是 coordinator 呢？回想 4.3.5.11 小节，不同 Consumer Group 的偏移量是保存在 Topic 为“__consumer_offsets”的日志里面的，每个具体的 Consumer Group 的偏移量保存在特定的分区里面，客户端如果想提交消费者的偏移量，则必须找到该消费组所在的分区，并向该分区的 Leader Replica 所在的 Broker Server 发送 OffsetCommitRequest 请求来保存偏移量，此 Broker Server 就称为该消费组的 coordinator。

那么高级消费者是如何查找某个特定消费组的 coordinator 呢？本质上就是向 Broker Server 发送 ConsumerMetadataRequest 请求，查询消费者的元数据，其 ConsumerMetadataResponse 里面会包含 coordinator，然后和该 coordinator 建立通信链路，其具体实现如下所示：

```
def channelToOffsetManager(
  group: String,
  zkClient: ZkClient,
  socketTimeoutMs: Int = 3000,
  retryBackOffMs: Int = 1000) = {
  // 从 /brokers/ids 中寻找在线的 Broker，然后和其建立通信链路
  var queryChannel = channelToAnyBroker(zkClient)
  var offsetManagerChannelOpt: Option[BlockingChannel] = None
```

```
//直到和 coordinator 建立通信链路为止
while (!offsetManagerChannelOpt.isDefined) {
  var coordinatorOpt: Option[Broker] = None
  //直到找到 coordinatorOpt 为止
  while (!coordinatorOpt.isDefined) {//
    try {
      //通信链路断开，则重新链接
      if (!queryChannel.isConnected)
        queryChannel = channelToAnyBroker(zkClient)
      //发送 ConsumerMetadataRequest 请求
      queryChannel.send(ConsumerMetadataRequest(group))
      //接收 ConsumerMetadataResponse
      val response = queryChannel.receive()
      val consumerMetadataResponse = ConsumerMetadataResponse.
        readFrom(response.buffer)
      if (consumerMetadataResponse.errorCode == ErrorMapping.NoError)
        //提取 consumerMetadataResponse 中的 coordinatorOpt
        coordinatorOpt = consumerMetadataResponse.coordinatorOpt
      else {
        //由 offsets.channel.backoff.ms 参数决定，默认为 1000ms，即休息 1s 重新循环
        Thread.sleep(retryBackOffMs)
      }
    }
    catch {
      case ioe: IOException =>
        queryChannel.disconnect()
    }
  }
  val coordinator = coordinatorOpt.get
  //coordinator 和 queryChannel 相同，直接返回 queryChannel
  if (coordinator.host == queryChannel.host && coordinator.port == queryChannel.port) {
    offsetManagerChannelOpt = Some(queryChannel)
  } else {
    //coordinator 和 queryChannel 不相同
    var offsetManagerChannel: BlockingChannel = null
    try {
      //链接 coordinator
      offsetManagerChannel = new BlockingChannel(
        coordinator.host,
        coordinator.port,
        BlockingChannel.UseDefaultBufferSize,
        BlockingChannel.UseDefaultBufferSize,
        socketTimeoutMs)
      offsetManagerChannel.connect()
      offsetManagerChannelOpt = Some(offsetManagerChannel)
      //关闭 queryChannel
      queryChannel.disconnect()
    }
    catch {
      case ioe: IOException =>
```

```
            if (offsetManagerChannel != null) offsetManagerChannel.disconnect()
            Thread.sleep(retryBackOffMs)
            offsetManagerChannelOpt = None
          }
        }
      }
      offsetManagerChannelOpt.get
    }
```

在高级消费者内部通过offsetsChannel保存和coordinator的链接。当auto.commit.enable配置为true的时候，高级消费者内部会自动间隔一定时间，将截止到当前时间的消费者消费的偏移量提交至Zookeeper或者Kafka，其间隔时间由auto.commit.interval.ms参数决定，默认为60×1000ms。提交至Zookeeper时会更新/consumers/[group]/offsets/[topic]/[partition]目录下的数据；提交至Kafka会利用offsetsChannel发送OffsetCommitRequest，其具体实现如下：

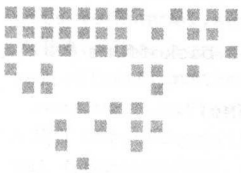
```
def commitOffsets(isAutoCommit: Boolean) {
  //计算重试次数，其中offsetsCommitMaxRetries由offsets.commit.max.retries决定，默认为5
  var retriesRemaining = 1 + (if (isAutoCommit) config.offsetsCommitMaxRetries else 0)
  var done = false
  while (!done) {
    val committed = offsetsChannelLock synchronized {
     //提取需要提交的偏移量(topic, partition, offset)
      val offsetsToCommit = immutable.Map(
        topicRegistry.flatMap { case (topic, partitionTopicInfos) =>
          partitionTopicInfos.map { case (partition, info) =>
            TopicAndPartition(info.topic, info.partitionId) -> OffsetAndMetadata(info.
              getConsumeOffset())
          }
      }.toSeq:_*)
      if (offsetsToCommit.size > 0) {
        //提交至Zookeeper
        if (config.offsetsStorage == "zookeeper") {
          //更新/consumers/[group]/offsets/[topic]/[partition]目录下的数据
          offsetsToCommit.foreach {
            case(topicAndPartition, offsetAndMetadata) =>
              commitOffsetToZooKeeper(topicAndPartition, offsetAndMetadata.offset)
          }
          true
        } else {
          //提交至Kafka，组装OffsetCommitRequest请求
          val offsetCommitRequest = OffsetCommitRequest(
            config.groupId,
            offsetsToCommit,
            clientId = config.clientId)
          //offsetsChannel保持链接
          ensureOffsetManagerConnected()
          try {
```

```
            // 发送 OffsetCommitRequest 请求
            offsetsChannel.send(offsetCommitRequest)
            // 获取 OffsetCommitResponse
            val offsetCommitResponse = OffsetCommitResponse.readFrom(offsetsChannel.
              receive().buffer)
            // 统计 OffsetCommitResponse 的信息
            val (commitFailed, retryableIfFailed, shouldRefreshCoordinator, errorCount)
              = {
              offsetCommitResponse.commitStatus.foldLeft(false, false, false, 0) {
               case(folded, (topicPartition, errorCode)) =>
                 if (errorCode == ErrorMapping.NoError && config.dualCommitEnabled) {
                   val offset = offsetsToCommit(topicPartition).offset
                   commitOffsetToZooKeeper(topicPartition, offset)
                 }
                 (folded._1 ||
                   errorCode != ErrorMapping.NoError,
                 folded._2 ||
                   (errorCode != ErrorMapping.NoError &&
                     errorCode != ErrorMapping.OffsetMetadataTooLargeCode),
                 folded._3 ||
                   errorCode == ErrorMapping.NotCoordinatorForConsumerCode ||
                   errorCode == ErrorMapping.ConsumerCoordinatorNotAvailableCode,
                 folded._4 + (if (errorCode != ErrorMapping.NoError) 1 else 0))
              }
            }
            // 如果 coordinator 异常，则断开
            if (shouldRefreshCoordinator) {
              offsetsChannel.disconnect()
            }
            // 提交失败，返回 false；否则成功
            if (commitFailed && retryableIfFailed)
              false
            else
              true
          }
          catch {
           // 异常情况下断开 offsetsChannel
            case t: Throwable =>
              offsetsChannel.disconnect()
              false
          }
        }
      } else {
        true
      }
    }
    // 判断是否完成
    done = if (isShuttingDown.get() && isAutoCommit) {
      retriesRemaining -= 1
      retriesRemaining == 0 || committed
```

```
      } else
        true
      // 如果没有完成，则休眠，其休眠时间由 offsets.channel.backoff.ms 参数决定，默认为 1000ms
      if (!done) {
        Thread.sleep(config.offsetsChannelBackoffMs)
      }
    }
  }
```

8.3 本章小结

本章讲解了客户端消费消息的内部实现原理，主要分为两种模式：简单消费者模式和高级消费者模式。针对这两种模式，Kafka 提供了两套消费者 API。这两者的本质区别在于如何管理偏移量，以及如何处理 Topic 状态的变化。如果使用者希望自己管理偏移量和自己处理消费过程中 Topic 的变化，则请使用简单消费者模式；如果使用者希望客户端 API 帮助使用者屏蔽偏移量的管量和消费过程中 Topic 的变化，则请使用高级消费者模式。



Chapter 9 第 9 章

Kafka 的典型应用

由于 Kafka 的高吞吐量、负载均衡、可扩展性等显著的特点，许多开源组件积极地把 Kafka 集成进各自的内部模块，进一步丰富了 Kafka 的生态圈。通过 Kafka 对外提供的生产者和消费者功能，可以无缝地打通 Kafka 和其他开源组件相互通信的数据通道。

Kafka 和其他开源组件的结合，使得可以对外提供更丰富的功能：Kafka 和 Storm 相结合，可以实时分析 Kafka 中的消息；Kafka 和 ELK 相结合，使得大规模的日志存储、分析、检索变得更加方便；Kafka 和 Hadoop 相结合，可以离线分析 Kafka 中的消息；Kafka 和 Spark 相结合，可以构建典型的在线数据分析平台。本章将介绍这些应用场景。

9.1 Kafka 和 Storm 的集成

Storm 是一个免费开源、分布式、高容错的实时计算系统。Storm 使得持续不断的流计算变得容易，弥补了 Hadoop 批处理所不能满足的实时要求。Storm 经常用于实时分析、在线机器学习、持续计算、分布式远程调用和 ETL 等领域。通过 Storm 中的 Spout 拉取 Kafka 消息，再通过 Storm 中的 Bolt 实现消息处理的业务逻辑，使得实时消息处理将变得更加简单、清晰、快捷。

9.1.1 Storm 简介

在 Storm 中，一个实时应用的计算任务被打包作为 Topology 发布，这同 Hadoop 的 MapReduce 任务相似。但是有一点不同的是：在 Hadoop 中，MapReduce 任务最终会执行完成后结束；而在 Storm 中，Topology 任务一旦提交后永远不会结束，除非你主动去停止任务。

计算任务 Topology 是由不同的 Spouts 和 Bolts，通过数据流连接起来的图。图 9-1 是一个 Topology 的结构示意图。

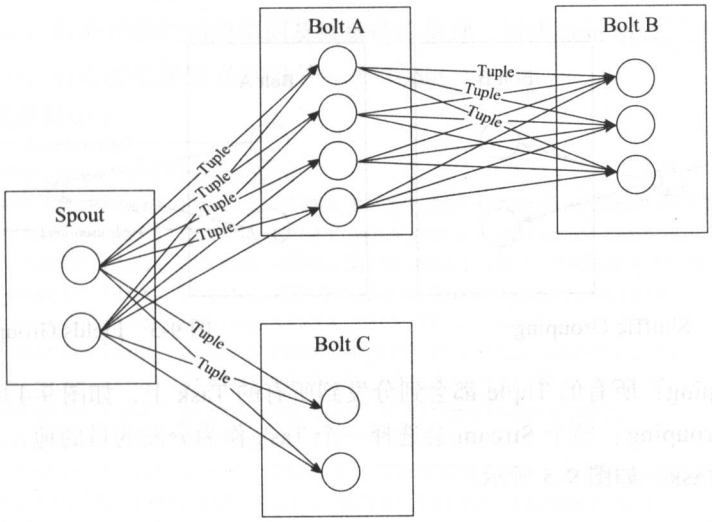


图 9-1　Storm 的 Topology

图 9-1 包含以下几个元素。

- Spout：Storm 中的消息源，用于为 Topology 生产消息（数据），一般是从外部数据源（如 Message Queue、RDBMS、NoSQL、Realtime Log）不间断地读取数据并发送给 Bolt。
- Bolt：Storm 中的消息处理者，对 Topology 的消息进行处理，Bolt 可以执行过滤、聚合、查询数据库等操作，而且可以一级一级地进行处理。
- Tuple：Storm 中消息传递的基本单元。

最终，Topology 会被提交到 Storm 集群中运行。

Topology 中每一个计算组件（Spout 和 Bolt）都有一个并行执行度，在创建 Topology 时可以指定，Storm 会在集群内分配对应并行度个数的线程来同时执行这一组件。那么，有一个问题：既然对于一个 Spout 或 Bolt，都会有多个 Task 线程来运行，那么如何在两个组件（Spout 和 Bolt）之间发送 tuple 元组呢？ Storm 提供了若干种数据流分发（Stream Grouping）策略用来解决这一问题。

目前 Storm 中提供了以下 7 种 Stream Grouping 策略：Shuffle Grouping、Fields Grouping、All Grouping、Global Grouping、Non Grouping、Direct Grouping、Local or Shuffle Grouping。

- Shuffle Grouping：Task 中的数据随机分配，可以保证每个同一级的 Bolt 上的 Task 处理的 Tuple 数量一致，如图 9-2 所示。

- Fields Grouping：根据 Tuple 中的某一个 Filed 或者多个 Filed 的值来划分。比如 Stream 根据 Field 为 user-id 来分发，相同 user-id 值的 Tuple 会被分发到相同的 Task 中，如图 9-3 所示。

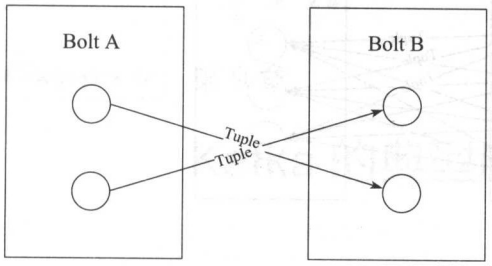


图 9-2 Shuffle Grouping

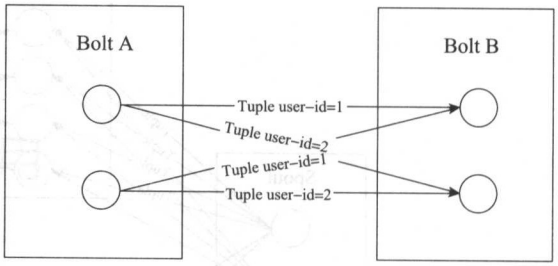


图 9-3 Fields Grouping

- All Grouping：所有的 Tuple 都会到分发到所有的 Task 上，如图 9-4 所示。
- Global Grouping：整个 Stream 会选择一个 Task 作为分发的目的地，通常是最新的那个 id 的 Task，如图 9-5 所示。

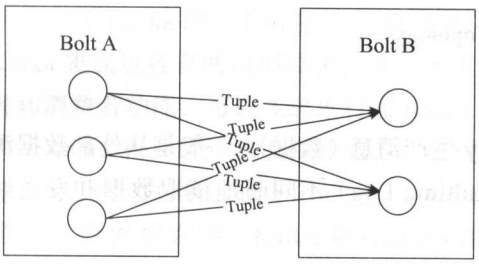


图 9-4 All Grouping

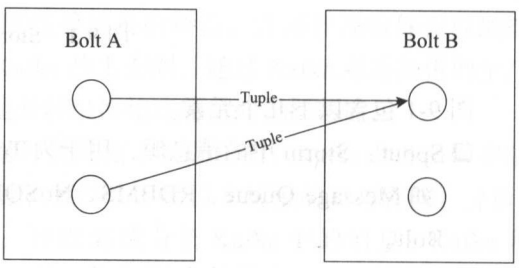


图 9-5 Global Grouping

- None Grouping：也就是你不关心如何在 Task 中做 Stream 的分发，目前等同于 Shuffle Grouping。
- Direct Grouping：产生数据的 Spout/Bolt 自己明确决定这个 Tuple 被 Bolt 的哪些 Task 所消费。
- Local or Shuffle Grouping：如果目标 Bolt 中的一个或者多个 Task 和当前产生数据的 Task 在同一个 JVM 进程里面，那么就走内部的线程间通信，将 Tuple 直接发给在当前 JVM 进程的目的 Task。否则，同 Shuffle Grouping 一样。

9.1.2 示例代码

假设利用 Spout 从 Kafka 消费消息，然后根据空格分词，统计单词数量，然后将当前输入的单词数量推送到另一个 Topic，则需要配置以下的 Spout 和 Bolt：

- KafkaSpout：负责从 Kafka 集群消费消息，并且需要定义消费数据的格式。

- SplitSentenceBolt：负责按照空格切分句子。
- WordCountBolt：负责对接收到的单词进行汇总统一，然后将单词“word”及其对应数量“count”向后传输。
- ReportBolt：负责对接收到的单词及数量进行整理，拼成 json 格式，然后继续向后传输
- KafkaBolt：负责将结果发送到另外一个 Topic。

其主要代码逻辑如下：

```
public class WordCountTopology {
  private static final String KAFKA_SPOUT_ID = "kafkaSpout";
  private static final String SPLIT_BOLT_ID = "splitSentenceBolt";
  private static final String WORD_COUNT_BOLT_ID = "wordCountBolt";
  private static final String REPORT_BOLT_ID = "reportBolt";
  private static final String KAFKA_BOLT_ID = "kafkabolt";
  private static final String CONSUME_TOPIC = "consumeTopic";
  private static final String PRODUCT_TOPIC = "productTopic";
  private static final String ZK_ROOT = "/topology/root";
  private static final String ZK_ID = "wordCount";
  private static final String DEFAULT_TOPOLOGY_NAME = "WordCountKafka";
  public static void main(String[] args) throws Exception {
    // 配置 Zookeeper 地址
    BrokerHosts brokerHosts =
      new ZkHosts("172.23.8.160:2181,172.23.8.161:2281,172.23.8.163:2381");
    // 配置 KafkaSpout 订阅的 Topic 等参数
    SpoutConfig spoutConfig = new SpoutConfig(brokerHosts, CONSUME_TOPIC,
      ZK_ROOT, ZK_ID);
    // 配置 KafkaSpout 消费 kafka 数据的处理流程
    spoutConfig.scheme = new SchemeAsMultiScheme(new MessageScheme());
    // 构造 Topology 构造器
    TopologyBuilder builder = new TopologyBuilder();
    // 设置 KafkaSpout
    builder.setSpout(KAFKA_SPOUT_ID, new KafkaSpout(spoutConfig));
    // 设置 SplitSentenceBolt
    builder.setBolt(SPLIT_BOLT_ID, new SplitSentenceBolt()).shuffleGrouping
      (KAFKA_SPOUT_ID);
    // 设置 WordCountBolt
    builder.setBolt(WORD_COUNT_BOLT_ID,
                    new WordCountBolt()).fieldsGrouping(SPLIT_BOLT_ID, new
                    Fields("word"));
    // 设置 ReportBolt
    builder.setBolt(REPORT_BOLT_ID, new ReportBolt()).shuffleGrouping(WORD_
      COUNT_BOLT_ID);
    // 设置 KafkaBolt
    builder.setBolt(KAFKA_BOLT_ID, new KafkaBolt<String, Long>())
      . shuffleGrouping(REPORT_BOLT_ID);
    Config config = new Config();
    Map<String, String> map = new HashMap<>();
    // 配置 Kafka broker 地址
    map.put("metadata.broker.list",
```

```
        "172.23.8.160:9092,172.23.8.161:9092,172.23.8.163:9092,172.23.8.164:9092");
    // serializer.class 为消息的序列化类
    map.put("serializer.class", "kafka.serializer.StringEncoder");
    // 配置 KafkaBolt 中的 kafka.broker.properties
    config.put("kafka.broker.properties", map);
    // 配置 KafkaBolt 生成的 topic
    config.put("topic", PRODUCT_TOPIC);
    if (args.length == 0) {
      // 本地模式提交
      LocalCluster cluster = new LocalCluster();
      cluster.submitTopology(DEFAULT_TOPOLOGY_NAME, config, builder.   createTopology());
      Utils.sleep(100000);
      cluster.killTopology(DEFAULT_TOPOLOGY_NAME);
      cluster.shutdown();
    } else {
      // 集群模式提交
      config.setNumWorkers(1);
      StormSubmitter.submitTopology(args[0], config, builder.createTopology());
    }
  }
}
```

其中 MessageScheme 的定义如下：

```
public class MessageScheme implements Scheme {
  @Override
  // 反序列化 Kafka 的消息
  public List<Object> deserialize(byte[] ser) {
    try {
      String msg = new String(ser, "UTF-8");
      return new Values(msg);
    } catch (UnsupportedEncodingException ignored) {
      return null;
    }
  }
  @Override
  // 定义 KafkaSpout 向后发送 tuple 的名字
  public Fields getOutputFields() {
    return new Fields("msg");
  }
}
```

SplitSentenceBolt 的定义如下：

```
public class SplitSentenceBolt extends BaseBasicBolt {
  @Override
  public void declareOutputFields(OutputFieldsDeclarer outputFieldsDeclarer) {
    outputFieldsDeclarer.declare(new Fields("word"));
  }
  @Override
  public void execute(Tuple input, BasicOutputCollector collector) {
```

```
    String sentence = input.getStringByField("msg");
    String[] words = sentence.split(" ");
    Arrays.asList(words).forEach(word -> collector.emit(new Values(word)));
  }
}
```

WordCountBolt 的定义如下：

```
public class WordCountBolt extends BaseBasicBolt {
  private Map<String, Long> counts = null;
  @Override
  public void prepare(Map stormConf, TopologyContext context) {
    this.counts = new ConcurrentHashMap<>();
    super.prepare(stormConf, context);
  }
  @Override
  public void declareOutputFields(OutputFieldsDeclarer outputFieldsDeclarer) {
    outputFieldsDeclarer.declare(new Fields("word", "count"));
  }
  @Override
  public void execute(Tuple input, BasicOutputCollector collector) {
    String word = input.getStringByField("word");
    Long count = this.counts.get(word);
    if (count == null) {
      count = 0L;
    }
    count++;
    this.counts.put(word, count);
    collector.emit(new Values(word, count));
  }
}
```

ReportBolt 的定义如下：

```
public class ReportBolt extends BaseBasicBolt {
  @Override
  public void execute(Tuple input, BasicOutputCollector collector) {
    String word = input.getStringByField("word");
    Long count = input.getLongByField("count");
    String reportMessage = "{'word': '" + word + "', 'count': '" + count + "'}";
    collector.emit(new Values(reportMessage));
  }
  @Override
  public void declareOutputFields(OutputFieldsDeclarer outputFieldsDeclarer) {
    outputFieldsDeclarer.declare(new Fields("message"));
  }
}
```

以上流程如图 9-6 所示。

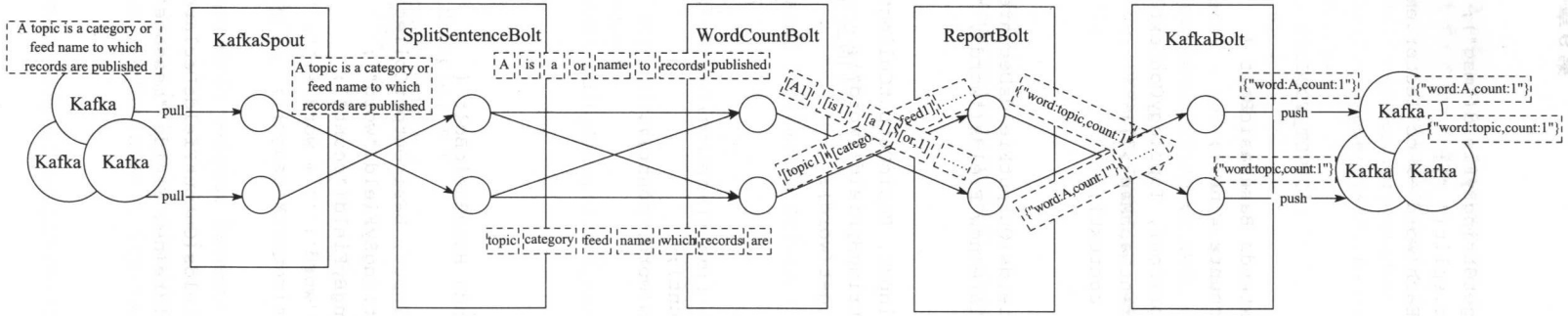


图 9-6　WordCountTopology 结构图

其中利用 KafkaSpout 从 Kafka 集群 pull 消息，并且通过 Shuffle Grouping 策略转发至 SplitSentenceBolt;SplitSentenceBolt 将接收到的消息按照空格切分成单词，并且通过 Fields Grouping 策略转发至 WordCountBolt;WordCountBolt 统计每个单词出现的次数，并且通过 Fields Grouping 策略转发至 ReportBolt;ReportBolt 将单词及其出现次数组装成 JSON 格式的字符串，并且通过 Fields Grouping 策略转发至 KafkaBolt;KafkaBolt 将 JSON 格式的字符串 push 到 Kafka 集群。

9.2　Kafka 和 ELK 的集成

日志主要包括系统日志、应用程序日志和安全日志。系统运维和开发人员可以通过日志了解服务器软硬件信息、检查配置过程中的错误及错误发生的原因。经常分析日志可以了解服务器的负荷、性能安全性，从而及时采取措施纠正错误。

通常，日志被分散地储存在不同的设备上。如果你管理上百台服务器，还在使用依次登录每台机器的传统方法查阅日志，是不是感觉很繁琐和效率低下。当务之急迫使我们使用集中化的日志管理，例如：开源的 syslog，将所有服务器上的日志收集汇总。

集中化管理日志后，日志的统计和检索又成为一件比较麻烦的事情，一般我们使用 grep、awk 和 wc 等 Linux 命令能实现检索和统计，但是对于要求更高的查询、排序和统计等要求，庞大的机器数量依然使用这样的方法难免有点力不从心。

开源实时日志分析平台 ELK（Elasticsearch，Logstash，Kibana）能够完美地解决我们上述的问题。通过 Logstash 从 Kafka 中获取用户提交的日志数据，然后将日志数据存储在 Elasticsearch 中，最后通过 Kibana 将日志数据进行可视化操作，使得一站式获取数据、存储数据、检索数据变得更加简洁、方便、迅速。

9.2.1　ELK 简介

ELK 其实并不是一款软件，而是一整套解决方案，是三个软件产品的首字母缩写，Elasticsearch、Logstash 和 Kibana。

Elasticsearch 是一个实时的分布式搜索和分析引擎，可用于全文搜索、结构化搜索以及分析。它是一个建立在全文搜索引擎 Apache Lucene 基础上的搜索引擎，使用 Java 语言编写。主要特点为：

- 实时分析。
- 分布式实时文件存储，并将每一个字段都编入索引。
- 文档导向，所有的对象全部是文档。
- 高可用性，易扩展，支持集群（Cluster）、分片和复制（Shards 和 Replicas）。
- 接口友好，支持 JSON。

Logstash 是一个具有实时渠道能力的数据收集引擎。使用 JRuby 语言编写。其作者是

世界著名的运维工程师乔丹·西塞 (Jordan Sissel)。主要特点为：

- 几乎可以访问任何数据。
- 可以和多种外部应用结合。
- 支持弹性扩展。

其功能如图 9-7 所示：

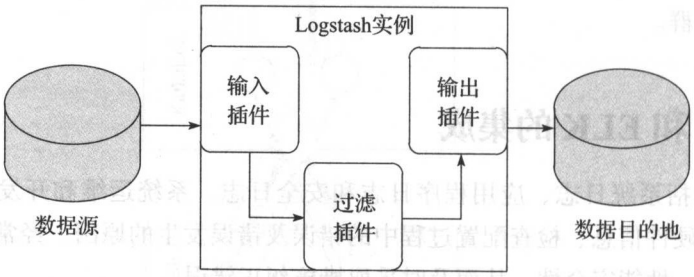


图 9-7 Logstash 功能

Logstash 负责从外部数据源加载数据，在内部进行过滤，然后输出到目的地存储。

Kibana 是一款基于 Apache 开源协议，使用 JavaScript 语言编写，为 Elasticsearch 提供分析和可视化的 Web 平台。它可以在 Elasticsearch 的索引中查找、交互数据，并生成各种维度的图表。

9.2.2 配置流程

当利用 Kafka 作为 Logstash 收集数据的源头，则此时 Logstash 起到收集 Kafka 数据，写入 Elasticsearch 的作用，其流程如图 9-8 所示。

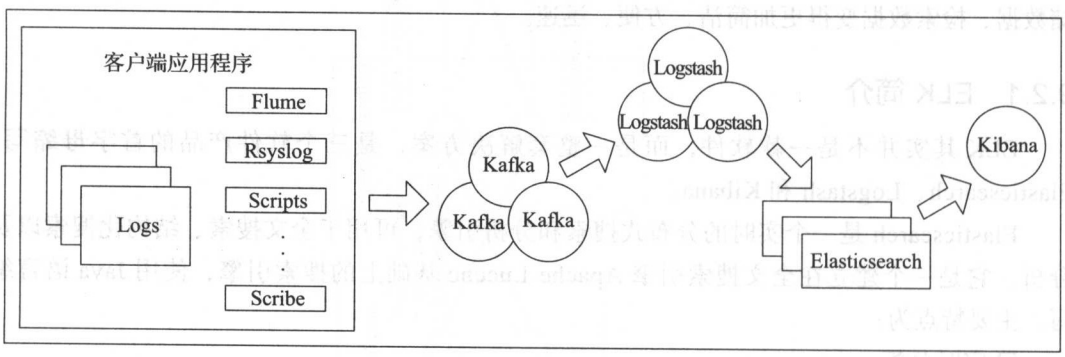


图 9-8 Kafka+ELK

其中客户端应用程序利用 Flume、Rsyslog、Scripts、Scribe 等方式将 Logs 发送到 Kafka，然后 Logstash 模块从 Kafka 拉取数据存放入 Elasticsearch，最后 Kibana 利用存放在 Elasticsearch 的数据进行可视化界面展示。因此以上流程的关键是如何配置 Logstash 消费 Kafka 的数据。Logstash 配置 Kafka 数据源的方法如下：修改位于 Logstash 安装目录下的

config 文件夹中的配置文件，需要配置以下两个参数：

- input：数据来源。
- output：数据存储到哪里。

具体实现代码如下：

```
input {
  # 输入 Kafka
  kafka {
    zk_connect => "172.23.8.160:2181"          # Zookeeper 地址
    topic_id => "mylog"                        # 消费的 topic 名称
  }
}
filter {
  #Only matched data are send to output.
}
output {
  // 输出 elasticsearch
  elasticsearch {
    action => "index"                          # ES 集群上的操作类型
    hosts  => "172.23.8.160:9200"              # ES 集群主机地址，可以是数组
    index  => "mylog"                          # ES 集群的索引名
  }
}
```

其他配置流程可参阅 ELK 相关书籍，此处不做过多的阐述。

9.3　Kafka 和 Hadoop 的集成

Hadoop 主要由分布式文件系统 HDFS 和分布式计算框架 MapReduce 组成。HDFS 提供了海量存储的能力，MapReduce 提供了分布式计算的能力。通过 MapReduce 可以将外部数据源的数据导入至 HDFS 内，也可以将内部 HDFS 内的数据导入至外部数据源。因此可以利用 MapReduce 将 Kafka 集群内部的数据经过初步分析导入至 HDFS 之内，也可以利用 MapReduce 将 HDFS 上的数据经过整理发送至 Kafka 集群。通过以上两种方式，可以实现 Kafka 和 HDFS 上的数据互通。

9.3.1　Hadoop 简介

Hadoop 是一个分布式系统基础架构，它实现了一个分布式文件系统（Hadoop Distributed File System，HDFS）。HDFS 有高容错性的特点，并且设计用来部署在低廉的硬件上；而且提供高吞吐量来访问应用程序的数据，适合那些有着超大数据集的应用程序。在 HDFS 的上一层是 MapReduce 引擎，采用分而治之的思想，把对大规模数据集操作的 Job，分发给一个主节点管理下的各个分节点共同完成，然后通过整合各个节点的中间结果，得到最终结果。其基本组成如图 9-9 所示。

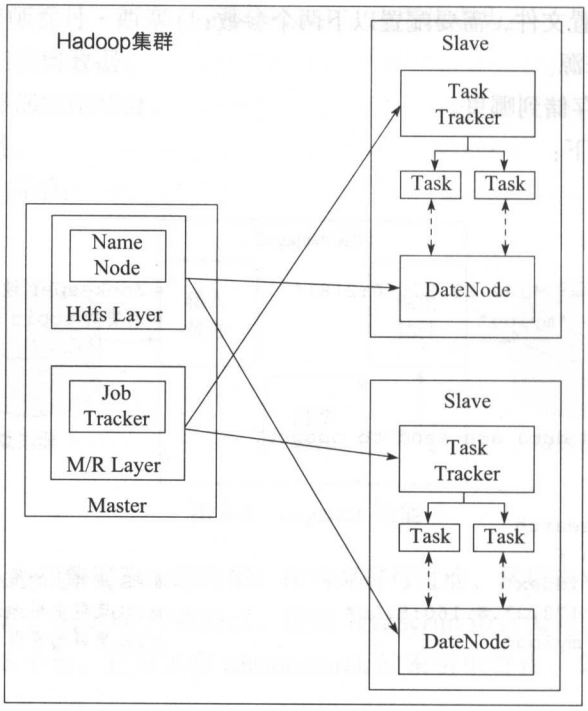


图 9-9　HDFS+MapReduce

其中主要包括以下组件：

- Name Node：这些节点存储文件块在 Data Node 节点中的分布信息。
- Data Node：这些节点存储文件块的真实数据。
- Job Tracker：这些节点负责将 MapReduce Job 分割成若干个 Task，并下发至 Task Tracker 节点。
- Task Tracker：这些节点负责执行 Job Tracker 下发的 Task。

MapReduce Job 的工作原理如图 9-10 所示。

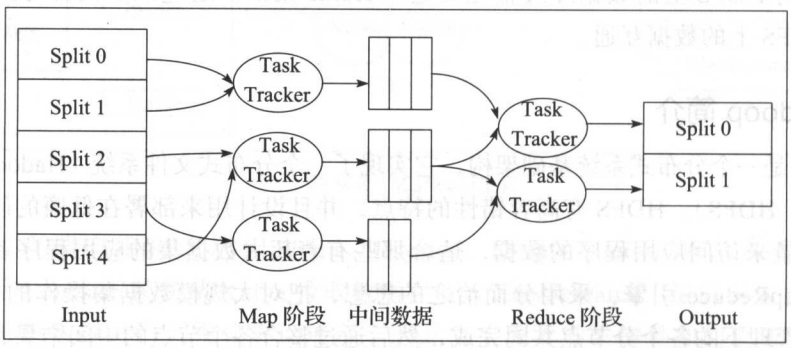


图 9-10　MapReduce Job 工作原理

Job 的运行由 Map 阶段和 Reduce 阶段组成，Map 阶段会输入 Input 数据（Input 数据会按照某种规则切分成若干个分片），然后输出中间数据，Reduce 阶段会汇总中间数据然后输出 Ouput 数据（Output 数据会按照某种规则切分成若干个分片）。

因此当 Input 数据来源于 HDFS，Output 数据流入 Kafka 时，则就是把 HDFS 上的数据导入到 Kafka，此时 Hadoop 充当的是生产者的角色，如图 9-11 所示。

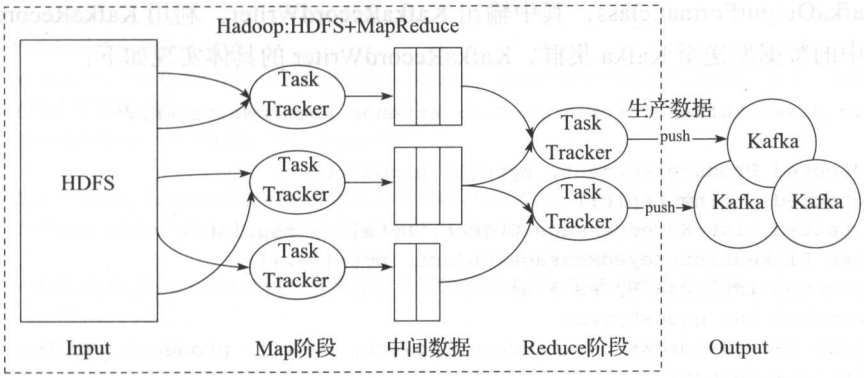


图 9-11　Hadoop-Producer

当 Input 数据来源于 Kafka，Output 数据流入 HDFS 时，就是把 Kafka 上的数据导入到 HDFS，此时 Hadoop 充当的是消费者的角色，如图 9-12 所示。

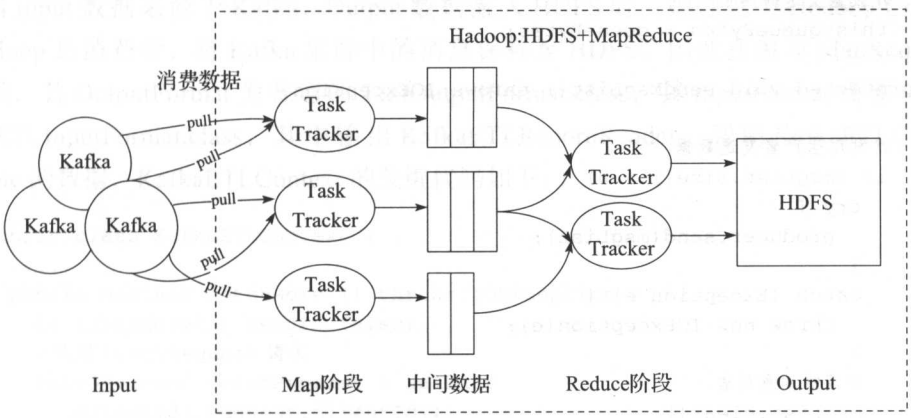


图 9-12　Hadoop-Consumer

9.3.2　示例代码

当 Hadoop 充当生产者角色时，本质上就是在 Reducer 阶段利用 Kafka 提供的 Producer 对象发送消息，并将其简称为 Hadoop-Producer；当 Hadoop 充当消费者角色时，本质上就是在 Map 阶段利用 Kafka 提供的 SimpleConsumer 对象消费消息，并将其简称为 Hadoop-

Consumer。接下来将就以上两种情况分别叙述其 MapReducer 编程时的关键代码。

9.3.2.1 Hadoop-Producer

当 Input 数据来源于 HDFS，Output 数据流入 Kafka 时，则称之为 Hadoop-Producer，即 Hadoop 是生产者，将 HDFS 上的数据发送至 Kafka 集群。因此在编写 MapReduce 程序的时候，其 InputFormatClass 为普通的 TextInputFormat.class，其 OutputFormatClass 需要自定义为 KafkaOutputFormat.class，其中输出 KafkaRecordWriter，利用 KafkaRecordWriter 将输出分片中的数据发送至 Kafka 集群，KafkaRecordWriter 的具体实现如下：

```
public class KafkaRecordWriter<K,V> extends RecordWriter<K,V>
{
  protected Producer<Object, byte[]> producer;
  protected String topic;
  protected List<KeyedMessage<Object, byte[]>> msgList =
    new LinkedList<KeyedMessage<Object, byte[]>>();
  protected int totalBytes = 0;
  protected int queueBytes;
  public KafkaRecordWriter(Producer<Object, byte[]> producer, String topic,
    int queueBytes)
  {
    //生产者
    this.producer = producer;
    //topic
    this.topic = topic;
    //批量发送的大小
    this.queueBytes = queueBytes;
  }
  protected void sendMsgList() throws IOException
  {
    //利用生产者发送数据
    if (msgList.size() > 0) {
      try {
        producer.send(msgList);
      }
      catch (Exception e) {
        throw new IOException(e);
      }
      //清除消息列表
      msgList.clear();
      totalBytes = 0;
    }
  }
  public void write(K key, V value) throws IOException, InterruptedException
  {
    byte[] valBytes;
    if (value instanceof byte[])
      valBytes = (byte[]) value;
    else if (value instanceof BytesWritable)
```

```
            valBytes = Arrays.copyOf(((BytesWritable) value).getBytes(), ((BytesWritable)
                                    value).getLength());
        else
            throw new IllegalArgumentException("KafkaRecordWriter expects byte array value
                                                to publish");
        // 如果已经达到批量发送的字节数或者消息个数已经超过32767，则先发送
        if ((totalBytes + valBytes.length) > queueBytes || msgList.size() >=
            Short.MAX_VALUE)
            sendMsgList();
        // 添加到待发送消息列表
        msgList.add(new KeyedMessage<Object, byte[]>(this.topic, key, valBytes));
        // 统计累计字节数
        totalBytes += valBytes.length;
    }
    public void close(TaskAttemptContext taskAttemptContext)
        throws IOException, InterruptedException
    {
        // 关闭KafkaRecordWriter时需要把剩余的待发送消息列表中的数据发送出去
        sendMsgList();
        producer.close();
    }
}
```

其他剩余代码可详细参考kafka源码包下的contrib/ hadoop-producer文件夹。

9.3.2.2 Hadoop-Consumer

当Input数据来源于Kafka，Output数据流入HDFS时，则称之为Hadoop-Consumer，即Hadoop是消费者，将Kafka集群中的消息保存至HDFS。因此在编写MapReduce程序的时候，其OutputFormat为普通的TextOutputFormat.class，其InputFormat需要自定义为KafkaETLInputFormat.class，其中输出KafkaETLRecordReader，利用KafkaETLContext消费Topic的数据，KafkaETLContext的关键代码如下：

```
public class KafkaETLContext {
    ......
    public boolean fetchMore () throws IOException {
        if (!hasMore()) return false;
        // 构建FetchRequest请求
        FetchRequest fetchRequest = builder
            .clientId(_request.clientId())
            .addFetch(_request.getTopic(), _request.getPartition(), _offset, _bufferSize)
            .build();
        long tempTime = System.currentTimeMillis();
        // 利用普通消费者发送FetchRequest
        _response = _consumer.fetch(fetchRequest);
        if(_response != null) {
            _respIterator = new ArrayList<ByteBufferMessageSet>(){{
                add((ByteBufferMessageSet) _response.messageSet(
                    _request.getTopic(), _request.getPartition()));
```

```
            }}.iterator();
        }
        _requestTime += (System.currentTimeMillis() - tempTime);
        return true;
    }
    ......
}
```

其他剩余代码可详细参考 kafka 源码包下的 contrib/ hadoop-consumer 文件夹。

9.4 Kafka 和 Spark 的集成

Spark 是一个围绕速度、易用性和复杂分析构建的大数据处理框架。与 MapReduce 流程相比，它主要通过两个方面来优化 Shuffle 过程：

- 利用多线程来执行具体的任务，而不是像 MR 那样采用进程模型，减少了任务的启动开稍；
- 中间数据保存在内存中，而不是像 MR 那样保存在 HDFS 上。

Spark 主要由以下几个模块构成：Spark-Core、Spark-Streaming、Spark-SQL、Spark-Graphx 和 Spark-ML。其中 Spark-Core 模块为其他模块提供计算引擎。Spark 可以和 Hadoop 完美结合，通过 Spark-Streaming 可以拉取 Kafka 中的数据，然后将其缓存在内存中，同时间隔一定时间将缓存中的数据刷新到 HDFS 文件系统上，最后利用 Spark-SQL 可以方便、快捷地分析位于内存和磁盘上的数据文件，这样就可以构成了一个小型的在线数据分析平台。

9.4.1 Spark 简介

Spark 集群主要由 Cluster Manager、Worker、Executor 和 Driver App 组成：

- Cluster Manager：Spark 的集群管理器，主要负责资源的分配与管理。集群管理器分配的资源属于一级分配，将各个 Worker 上的内存、CPU 等资源分配给应用程序，但是并不负责对 Executor 的资源分配。目前，Standalone、YARN、Mesos、EC2 等都可以作为 Spark 的集群管理器。
- Worker：Spark 的工作节点。对 Spark 应用程序来说，由集群管理器分配得到资源的 Worker 节点主要负责以下工作：创建 Executor，将资源和任务进一步分配给 Executor，同步资源信息给 Cluster Manager。
- Executor：执行计算任务的一组进程。主要负责任务的执行以及与 Worker、Driver App 的信息同步。
- Driver App：客户端驱动程序，也可以理解为客户端应用程序，用于将任务程序提交到 Spark，并与 Cluster Manager 进行通信与调度。

其基本架构图如图 9-13 所示。

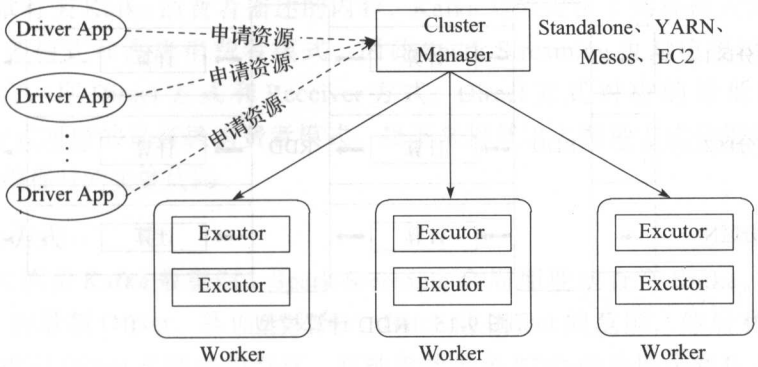


图 9-13　Spark 基本架构图

整个 Spark 主要由以下模块组成：

- Spark-Core：Spark 的核心功能实现，例如部署模式、存储体系、任务提交与执行、计算引擎等。
- Spark-SQL：提供 SQL 处理能力，便于熟悉关系型数据库操作的工程师进行交互查询。此外，还为熟悉 Hadoop 的用户提供 Hive SQL 处理能力。
- Spark-Streaming：提供流式计算处理能力，目前支持 Kafka、Flume、Twitter、MQTT、ZeroMQ、Kinesis 和简单的 TCP 套接字等数据源。此外，还提供窗口操作。
- Spark-Graphx：提供图计算处理能力。
- Spark-ML：提供机器学习相关的统计、分类、回归等领域的多种算法实现。

Spark-SQL、Spark-Streaming、Spark-Graphx、Spark-ML 的能力都是建立在核心引擎 Spark-Core 之上，如图 9-14 所示。

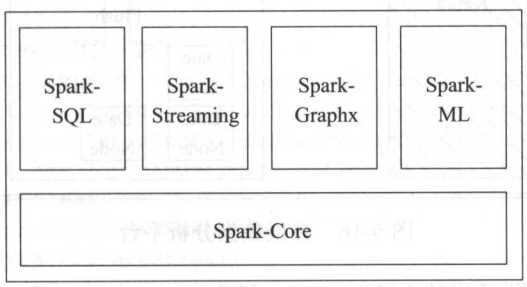


图 9-14　Spark 各模块依赖关系

在整个 Spark 的计算引擎中，任务的计算通过 RDD 来表示。RDD 可以看做是对各种数据计算模型的统一抽象，Spark 的计算过程主要是 RDD 的迭代计算过程，如图 9-15 所示。RDD 的迭代计算过程非常类似于管道。分区数量取决于 partition 数量的设定，每个分区的

数据只会在一个 Task 中计算。所有分区可以在多个机器节点的 Executor 上并行执行。

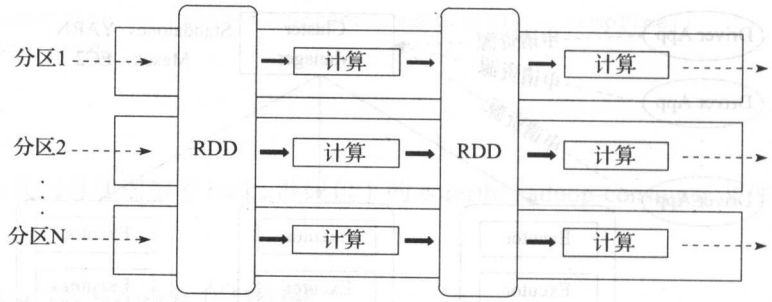


图 9-15 RDD 计算模型

由于 RDD 的输入可以是 Kafka、Flume、Twitter、MQTT、ZeroMQ 等数据源，RDD 的输出可以是内存、HDFS、HBase 等数据源，因此利用 Spark-Streaming 拉取 Kafka，然后持久化至 Spark 的内存，后台通过定时刷新线程将 Spark 内存的数据持久化至 HDFS，对外通过 Spark-SQL 提供 SQL 访问接口，这样就构成了一个典型的在线数据分析平台，其架构图如图 9-16 所示。

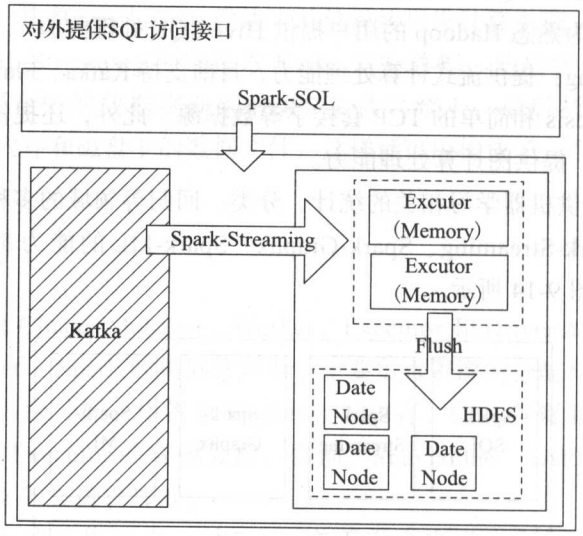


图 9-16 在线数据分析平台

Spark-Streaming 提供了两种方式和 Kafka 结合：

- Direct 方式，底层对应的是 Kafka 的低级消费者。
- Receiver 方式，底层对应的是 Kafka 的高级消费者。接下来主要就这两种方式的客户端编程做简要阐述，不会深入内部的实现原理。如果想深入了解内部的实现原理，可以参考 Spark 相关书籍。

9.4.2 示例代码

回顾第8章有关Kafka消费者描述的内容，Kafka对外提供了两种模式消费数据，分别为低级消费者模式和高级消费者模式，因此Spark-Streaming也对应提供了两种方式和Kafka相结合，即Direct方式和Receiver方式。Direct方式对应的是低级消费者模式，Receiver方式对应的是高级消费者模式。接下来将就以上两种方式分别叙述其Spark-Streaming流式编程时的关键代码。

9.4.2.1 Direct方式

Direct方式消费Kafka数据时，Spark-Sreaming会周期性地查询Kafka，来获得每个Topic+Partition的最新Offset，从而定义每个Batch的Offset的范围，然后利用简单消费者API去获取指定Offset范围内的数据，其抽象数据集RDD的分区个数取决于Kafka的Topic的分区个数，且前后两者分区一一对应，并且客户端需要持久化偏移量。其客户端的简要代码如下：

```
public class Direct implements Serializable {
  public static void main(String[] args) {
    SparkConf conf = new SparkConf().setAppName("Direct");
    //Spark的Cluster Manager地址，当前采用Standalone方式
    String sparkMaster = "spark://172-23-8-160:7077,172-23-8-161:7077";
    conf.setMaster(sparkMaster);
    //Spark上下文
    JavaSparkContext javaSparkContext = new JavaSparkContext(conf);
    //Spark-Streaming上下文
    JavaStreamingContext javaStreamingContext =
      new JavaStreamingContext(javaSparkContext, new Duration(10000));
    //设置Direct的参数
    Map<String, String> kafkaParams = new HashMap<String, String>();
    //设置Kafka的Broker列表
    kafkaParams.put("metadata.broker.list",
      "172.23.8.160:9092,172.23.8.161:9092,172.23.8.162:9092");
    //设置需要消费的Topic的分区详情
    Map<TopicAndPartition, Long> fromOffsets = new HashMap<TopicAndPartition, Long>();
    fromOffsets.put(new TopicAndPartition("Topic", 0), 0L);
    fromOffsets.put(new TopicAndPartition("Topic", 1), 0L);
    fromOffsets.put(new TopicAndPartition("Topic", 2), 0L);
    //申请JavaInputDStream
    JavaInputDStream directKafkaStream =
      KafkaUtils.createDirectStream(
        javaStreamingContext,
        String.class,
        String.class,
        StringDecoder.class,
        StringDecoder.class,
        String.class,
        kafkaParams,
```

```
            fromOffsets,
            new Function<MessageAndMetadata<String, String>, String>() {
              @Override
              public String call(MessageAndMetadata<String, String> v1) throws Exception {
                return v1.message();
              }
            });
        //针对每次任务的 RDD 进行处理
        directKafkaStream.foreachRDD(new Function<JavaRDD<String>, Void>() {
          @Override
          public Void call(JavaRDD<String> batchRdd) throws Exception {
            try {
              //持久化至内存
            } catch (Exception e) {
              e.printStackTrace();
            }
            return null;
          }
        });
        //启动 SparkStreaming
        javaStreamingContext.start();
        //等待
        javaStreamingContext.awaitTermination();
      }
    }
```

9.4.2.2 Receiver 方式

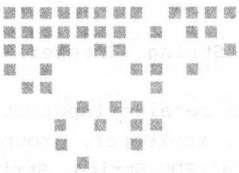
Receiver 是使用 Kafka 的高层次 Consumer API 来实现的，在 Excutor 上会常住若干个 Receiver 线程，其间隔一定较短的时间会将 Topic 的所有分区数据作为抽象数据集 RDD 的某个分区数据，间隔一定较长的时间会形成当前 RDD 的所有分区数据，其抽象数据 RDD 的分区数据不和原始 Kafka 的 Topic 的分区数据一一对应。Receiver 方式内部会记录每次消费的偏移量，客户端不需要持久化该偏移量。其客户端的简要代码如下：

```
public class Receiver implements Serializable {
  public static void main(String[] args) {
    SparkConf conf = new SparkConf().setAppName("Receiver");
    //Spark 的 Cluster Manager 地址，当前采用 Standalone 方式
    String sparkMaster = "spark://172-23-8-160:7077,172-23-8-161:7077";
    conf.setMaster(sparkMaster);
    //Spark 上下文
    JavaSparkContext javaSparkContext = new JavaSparkContext(conf);
    //Spark-Streaming 上下文
    JavaStreamingContext javaStreamingContext =
      new JavaStreamingContext(javaSparkContext, new Duration(10000));
    //Kafka 集群的 Zookeeper 地址
    String zookeeper = "172.23.8.160:2181,172.23.8.161:2181,172.23.8.162:
      2181/kafka";
    //消费组
```

```
        String group = "test-consumer-group";
        //Topic对应的Receiver个数
        Map<String, Integer> topicMap = new HashMap<String, Integer>();
        topicMap.put("Topic", 3);
        JavaPairReceiverInputDStream<String, String> receiverInputDStream =
        KafkaUtils.createStream(javaStreamingContext, zookeeper, group, topicMap);
        receiverInputDStream.foreach(new Function<JavaPairRDD<String, String>, Void>() {
          @Override
          public Void call(JavaPairRDD<String, String> batchRdd) throws Exception {
            //持久化至内存
            return null;
          }
        });
        //启动SparkStreaming
        javaStreamingContext.start();
        //等待
        javaStreamingContext.awaitTermination();
      }
    }
```

9.5 本章小结

本章简要讲述了Kafka和其他开源组件的典型集成，对不同的集成方式做了简单的介绍，但是没有深入内部详细的实现细节，而是停留在客户端代码的编写上，主要是想以此为引子激发读者的兴趣。至于更深层次的内部实现原理，希望读者自己查阅相关开源组件的资料去进一步深入了解。



Chapter 10 第 10 章

Kafka 的综合实例

第 9 章主要描述了 Kafka 和大数据相关的典型开源组件集成，为了更好地说明 Kafka 在真实业务场景中的应用，本章将介绍一个实例，描述 Kafka 在安防行业里的运用。由于安防监控行业随着平安城市、智慧城市和智能交通等大型安防项目的发展，移动互联设备激增，产生了海量的非结构化图片视频数据，带动了大数据相关的存储、管理和分析等应用，其中消息系统 Kafka 作为数据总线扮演着至关重要的角色。

10.1 安防大数据的主要应用

安防大数据的应用主要体现在以下两个方面：智能交通和公安执法。

前者主要指：针对交通行业的海量数据处理需求，智能交通管理系统可以在海量数据、恶劣网络环境和复杂业务处理情况下，实现大量图片、车辆数据、视频数据的网络传输和快速持久化存储，同时对任意站点的图像进行显示，对任意站点的视频进行流畅播放、实时进行比对报警，快速进行多条件检索，并且将各类多媒体数据和车辆数据合二为一。系统实现对目前的城市道路交通中异常行为的智能识别和自动报警等，从而减轻了交管监控人员的工作负担，提高了监测的准确度，使得交通管理工作更高效。比如，实时交通状况分析可通过视频实时分析道路交通流量，然后综合分析统计出全城市的交通状况；套牌分析可通过视频进行车牌识别，按照一定的规则 (如最近抓拍记录的时间间隔和距离间隔) 在全城市中检索相同车牌的汽车。在上述场景中会产生大量的车辆抓拍数据、实时监控视频流和历史监控视频流，它们都需要持久化存储下来。

后者主要指：犯罪嫌疑人追查，可通过输入嫌疑人照片进行人脸特征识别并在所有视

频中寻找该人脸；犯罪嫌疑车辆追查，可输入嫌疑车的照片或颜色车型等相关特征在所有视频中寻找；人车物的轨迹分析，即在所有视频中按照特征查找指定的人车物并绘制其时空轨迹等。在上述场景会利用人脸抓拍数据、车辆抓拍数据、监控视频流来协助破案。

上述提到的车辆图片、人脸图片、监控视频流都属于非结构化数据，都需要转化为结构化数据，并且持久化至分布式数据库中，同时非结构化原始数据也需要持久化至分布式存储中。除了这些非结构化数据，还涉及一些结构化数据的统计和布控，例如实时交通状况上报需要收集某个车道的实时交通流量信息；布控套牌车需要明确指定嫌疑套牌车车牌号，实时显示该套牌车的行车轨迹等。

在上述数据的流转中都需要利用消息系统 Kafka。Kafka 可以解耦数据的生产者和数据的消费者之间的强耦合，并且它作为数据总线使得数据的生产者和数据的消费者只需要和它本身交互即可，而两者之间无须直接交互，由此大大降低了各种业务对接的复杂度。

10.2　Kafka 在安防整体解决方案中的角色

在 10.1 小节中提到：Kafka 可以作为安防大数据中的数据总线，数据的生产者和数据的消费者只需要实现基于数据的接口层，并且两边遵守相同的接口约束即可。因此通过 Kafka 这个消息系统中间件可以将非结构化数据（图片、视频）和结构化数据（布控、统计）与大数据相关的组件（分布式数据库、分布式存储、分布式计算框架等）隔离起来，然后通过具体的业务组合相关的组件，其典型的架构如图 10-1 所示。

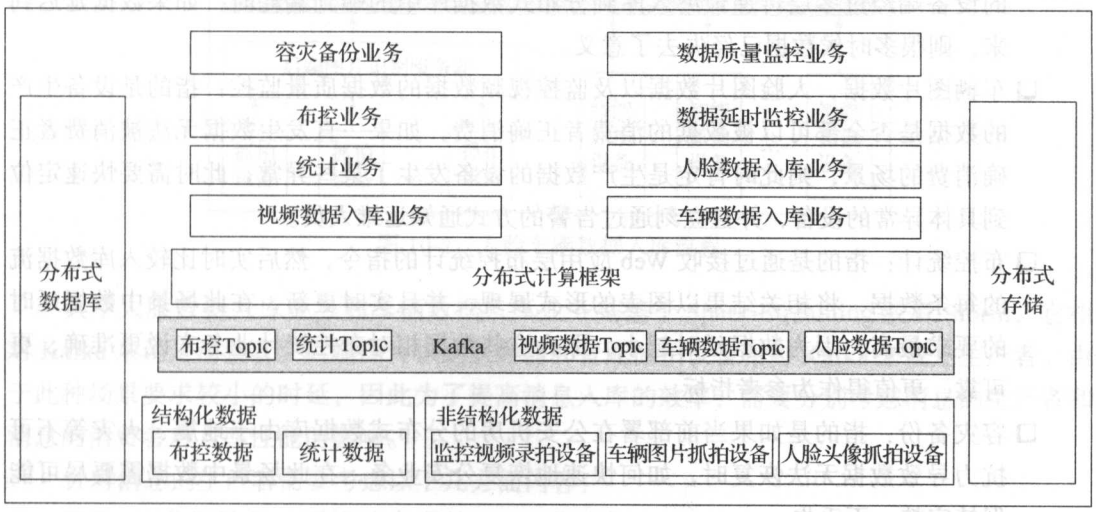


图 10-1　典型架构

图中的分布式数据库指的是 NoSQL 数据库，例如 Hbase、Redis、LevelDB 等；分布式计算框架包括 Spark、MapReduce 等；分布式存储包括 HDFS、GFS、TFS 等。结构化数据

是通过 Web 界面操作人员下发产生的，非结构化数据是通过车辆图片抓拍设备、人脸头像抓拍设备以及其他监控视频抓拍设备产生的。位于分布式计算框架之上的各种业务需要利用分布式数据库、分布式存储和 Kafka 中的相关数据，并且该系统的所有初始数据都需要经 Kafka 消息系统中间件这个模块流转。

10.3 典型业务

在基于 Kafka 作为数据总线的安防整体解决方案中，一般涉及以下典型业务：车辆图片数据和人脸图片数据的入库，监控视频数据的入库，车辆图片数据和人脸图片数据的数据时效性监控，车辆图片数据、人脸图片数据以及监控视频数据的数据质量监控，布控统计以及容灾备份等。其业务的具体说明如下：

- 车辆图片数据和人脸图片数据的入库：指的是车辆图片和人脸图片等非结构化数据如何及时快速地转化为结构化数据，然后通过 Kafka 消息队列进入到分布式数据库中，在此场景中数据延时需要尽可能小。
- 监控视频数据的入库：指的是如何准确分析出监控视频中的移动目标（人、车、物），并且提取关键移动目标的关键特征将其转化为结构化数据，然后通过 Kafka 消息队列进入到分布式数据库中，在此场景中数据分析需要尽可能准确，数据延时要求没有像车辆图片和人脸图片的场景需求那么严格。
- 车辆图片数据和人脸图片数据的数据时效性监控：指的是如何监测数据从生产数据的设备端经过多层传递最终入库到分布式数据库中的端到端耗时，如果数据延迟到来，则很多时候数据已经失去了意义。
- 车辆图片数据，人脸图片数据以及监控视频数据的数据质量监控：指的是设备生产的数据是否全部可以被数据的消费者正确消费，如果一直发生数据无法被消费者正确消费的场景，则此时肯定是生产数据的设备发生了某些异常，此时需要快速定位到具体异常的设备，并且立刻通过告警的方式通知运维人员。
- 布控统计：指的是通过接收 Web 应用层布控统计的指令，然后实时比较入库数据流的每条数据，将相关结果以图表的形式展现，并且实时更新，在此场景中数据延时的要求最高，因为数据延时越小，那么这些数据相对布控统计业务来说更准确，更可靠，更值得作为参考指标。
- 容灾备份：指的是如果当前部署在公安机房的分布式数据库由于地震、火灾等不可抗力导致数据无法恢复时，如何快速地恢复公安业务，在此场景中数据需要尽可能保持完整，不丢失。

下面将围绕以上业务阐述内部具体的实现原理，不仅需要设计 Kafka 和分布式计算框架、分布式存储以及分布式数据库这三者之间的联系，也需要特殊设计 Kafka 本身的消息格式。

10.3.1　车辆人脸图片数据的入库

车辆和图片数据本身属于非结构化数据，需要将其转化为结构化数据存入分布式数据库中。其中转化为结构化数据的步骤一般在两个地方实现：

1）后端图片识别服务器：针对前端设备比较老旧，智能化水平不高的场景。

2）前端智能化抓拍设备：针对智能化水平较高的前端设备。

当后端图片识别服务器承担非结构化数据转化为结构化数据时，需要汇总多个前端设备的图片，然后快速分析，最后将结构化数据发送至 Kafka 消息队列；当前端智能化抓拍设备承担非结构化数据转化为结构化数据时，可以直接将结构化数据发送至 Kafka 消息队列。

人脸数据入库业务和车辆数据入库业务会负责从 Kafka 消息队列将结构化数据转移至分布式数据库中。由于考虑到此种数据入库场景要求较小的延时，一般会采用分布式计算框架中的流式计算引擎，比如 Spark-Streaming 或者 Storm，其典型的架构如图 10-2 所示。

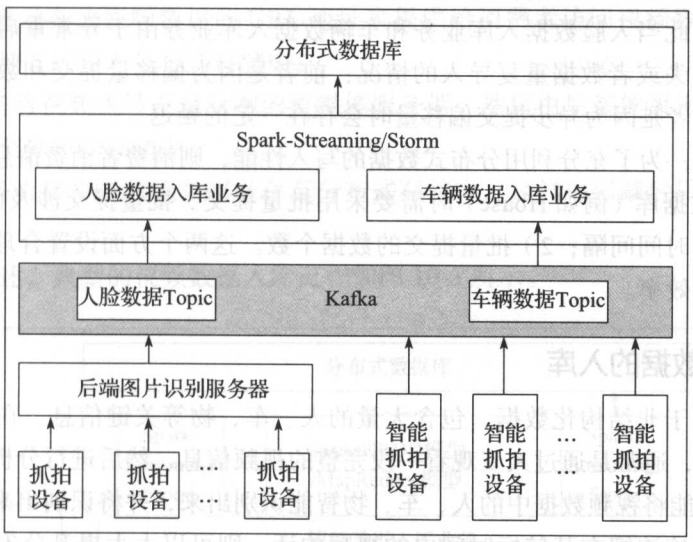


图 10-2　人脸车辆数据入库流程

其中人脸数据入库业务和车辆数据入库业务是基于 Spark-Streaming 或者 Storm，它相对 Kafka 来说是消费者。后端图片识别服务器和智能抓拍设备相对 Kafka 来说是生产者。由于此种场景要求较小的时延，因此为了提高消息入库的效率，需要分别考虑消息的生产者和消息的消费者对入库性能的影响。

针对消息的生产者需要考虑以下几方面内容：

- Topic 分区数目：要设计合理的 Topic 分区数目。Topic 分区个数太少无法充分利用磁盘的性能，最终会导致 Kafka 集群整体的吞吐量降低；Topic 分区个数太多会增加 Kafka 内部管理 Topic 的资源开销（尤其是 Kafka 和 Zookeeper 的通信交互），最终导致 Kafka 集群的稳定性降低。

- 数据倾斜：Kafka 中的消息是根据消息的分区键决定消息最终路由的分区，如果消息的分区键设计不合理，导致大量的消息都发往相同的分区，则会产生数据倾斜，此时会加重消息消费者的负担，最终会影响消息入库的延时。
- 发送模式：同步模式发送消息时无法保证每次发送的数据量是否合理，因此每次发送无法做到最优，而异步模式发送数据时可以将发送过程全部委托为生产者客户端后台管理，应用层无需主动触发发送。因此相比较异步模式发送消息和同步模式发送消息，前者可以带来更大的吞吐量。

针对消息的消费者需要考虑以下几个方面内容：

- 偏移量持久化机制：当利用流式计算框架消费消息时，需要考虑 Topic 偏移量的持久化机制。为了严格防止数据丢失，建议使用简单消费者模式，因为偏移量是客户端自己管理，从而可以在消息真正入库的时候持久化偏移量。如果使用高级消费者模式，则偏移量是客户端提交线程异步自动提交的，无法做到数据入库之后再提交偏移量，因此当人脸数据入库业务和车辆数据入库业务由于异常重启重新导入时会发生数据丢失或者数据重复导入的情况：前者是因为偏移量提交和数据入库无法真正同步，后者是因为异步提交偏移量时会存在一定的延迟。
- 批量入库：为了充分利用分布式数据的写入性能，则消费者消费消息之后将消息存入分布式数据库（例如 Hbase）时需要采用批量提交。批量提交涉及两个方面：1）批量提交的时间间隔；2）批量提交的数据个数。这两个方面设置合理会提升消息消费者的工作效率。

10.3.2 视频数据的入库

视频数据属于非结构化数据，包含大量的人、车、物等关键信息。在传统的公安和交警办案的过程中，通常是通过人工观看一段完整的视频信息，然后进行分析，此种办案方式效率低下。如果能将视频数据中的人、车、物智能识别出来，并将识别出来的信息在某长段时间内的行为轨迹浓缩在某较短时间内的视频的话，则可以大大提升公安和交警办案的效率，这就是所谓的浓缩摘要。

视频数据入库就是通过对视频数据进行浓缩摘要，将视频数据中的人、车、物转化为结构化数据，并且浓缩以上视频中关键物体的行为轨迹。此种场景属于事后分析，对于数据入库的延时并不是很严格，主要是需要准确分析出视频数据中的关键信息。

由于数据入库的时效性要求不高，并且视频分析需要消耗大量的计算资源，因此可以基于分布式计算框架中的批处理模型，例如基于 Spark 的 RDD 模型或者基于 Hadoop 中的 MapReduce 模型，并且以上两种批处理模型目前正在和 GPU（图形处理器）慢慢融合，充分利用 GPU 和 CPU 可以大大加速视频分析的速度和准确度。

那么 Kafka 在视频数据入库中扮演什么角色呢？在视频数据入库中，Kafka 主要流转两种类型的数据：1）控制流；2）数据流。控制流指的是 Web 端办案人员下发分析某段视频

的指令至 Kafka 中，此指令中不包含视频的真实数据，而是包含视频的 URL 地址，视频的真实数据是存放在分布式存储中的，通过 URL 地址可以访问视频的真实数据；数据流指的是批处理的时候视频数据入库业务从 Kafka 中拉取控制流，根据 URL 地址加载真实的视频数据，然后利用分布式计算框架将非结构化数据转化为结构化数据，最后将结构化数据发送到 Kafka 中。一旦结构化数据落入到 Kafka 的视频数据 Topic 中，则视频数据入库业务又从其订阅的视频数据 Topic 中拉取结构化数据存入分布式数据库中。由于 Kafka 充当的是数据总线的作用，因此视频数据入库业务只需要针对不同的 Topic 进行不同的处理：

1）如果是控制流 Topic，则下发视频浓缩摘要的任务至分布式计算框架上。

2）如果是数据流 Topic，则拉取结构化数据至分布式数据库中。

除此之外需要说明的是 Kafka 中不适合存储真实的视频数据，即非结构化数据。非结构化数据占用空间大，如果该数据存储在消息中不仅会影响客户端消息发送的吞吐量，而且会严重占用 Kafka 集群服务端的磁盘空间，导致影响 Kafka 集群其他 Topic 的使用。因此 Kafka 的消息中通过存放视频数据的 URL 地址来提供给消费者访问视频数据的能力。视频数据一般通过两种方式写入分布式存储中：

1）抓拍设备将视频流转发至后端的流媒体服务器，然后由后端的流媒体服务器统一将视频流数据保存至分布式存储中。

2）抓拍设备直接将视频流数据保存至分布式存储中，这样可以减轻后端流媒体服务器的写入压力。

综合以上所述，典型的视频数据入库流程如图 10-3 所示。

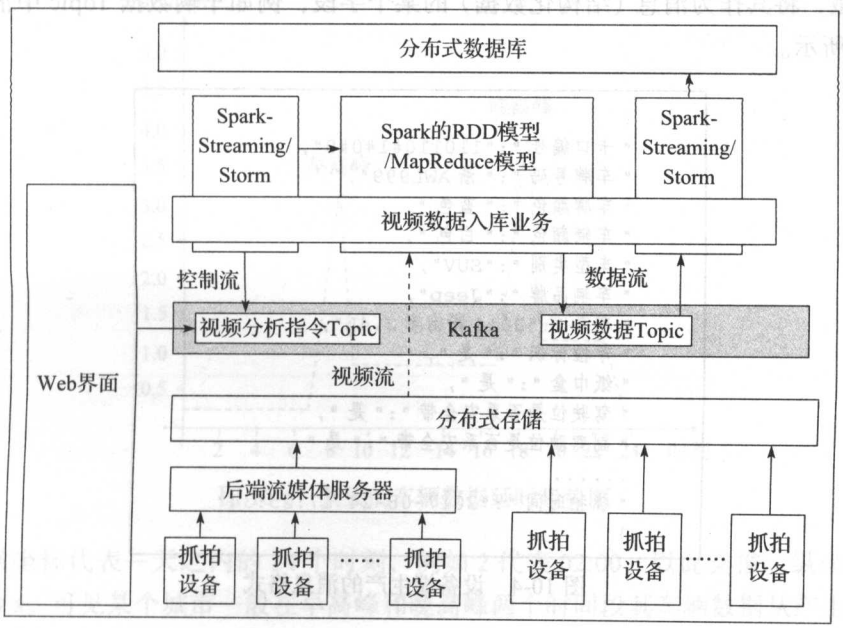


图 10-3　视频数据入库流程

说明如下：

1）流媒体服务器或者抓拍设备都可以将视频流写入分布式存储中。

2）办案人员通过 Web 界面下发视频分析指令至 Kafka 中的视频分析指令 Topic。

3）视频数据入库业务通过 Spark-Streaming 或者 Storm 消费视频分析指令 Topic 中的消息，然后根据消息生成批处理任务。

4）批处理任务利用 Spark 的 RDD 或者 Hadoop 的 MapReduce 从分布式存储中拉取视频流进行分布式计算，然后将计算结果（结构化数据）发送至视频数据 Topic。

5）视频数据入库业务通过 Spark-Streaming 或者 Storm 消费视频数据 Topic 中的结构化数据，然后存入分布式数据库。

10.3.3 数据延时的监控

数据延时指的是车辆图片数据和人脸图片数据（非结构化数据）从设备端产生时刻开始，经图像识别算法转化为结构化数据，然后将结构化数据存入分布式数据库中的端到端耗时。如果该端到端耗时比较久，则针对某些实时要求很高的场景而言，此时数据是无效的，不能起到数据本身的意义，比如交警通过车辆图片数据实时跟踪某辆车的行车轨迹，公安通过人脸图片数据实时跟踪某个人的移动轨迹等。因此需要监测整体的数据延时情况，一旦超过某个阈值，需要通过电话、短信等手段通知运维人员。

那么如何跟踪从非结构化数据产生到结构化数据进入分布式数据库的时间呢？通常的方法是通过特殊设计消息格式来达到监测数据延时的功能。首先记录设备端产生非结构化数据的时间戳，将其作为消息（结构化数据）的某个字段，例如车辆数据 Topic 中消息的格式如图 10-4 所示。

```
{
"卡口编号":"110110#1#0#0",
"车牌号码":"浙AWL999",
"车牌颜色":"蓝色",
"车辆颜色":"白色",
"车型类别":"SUV",
"车辆品牌":"Jeep",
"车系":"2014指南者2.4L",
"年检标识":"是",
"纸巾盒":"是",
"驾驶位是否系安全带":"是",
"副驾驶位是否系安全带":"是",
......
"抓拍时间":"2017-08-24 17:25:34"
}
```

图 10-4 设备端生产的消息格式

其中抓拍时间代表的就是车辆数据产生的时间。其次车辆数据入库业务通过 Spark-

Streaming 或者 Storm 消费车辆数据 Topic 之后，将结构化数据存入分布式数据库之前针对该结构化数据新增一个入库时间字段，记录当前的时刻，如图 10-5 所示。

```
{
"卡口编号":"110110#1#0#0",
"车牌号码":"浙AWL999",
"车牌颜色":"蓝色",
"车辆颜色":"白色",
"车型类别":"SUV",
"车辆品牌":"Jeep",
"车系":"2014 指南者2.4L",
"年检标识":"是",
"纸巾盒":"是",
"驾驶位是否系安全带":"是",
"副驾驶位是否系安全带":"是",
......
"抓拍时间":"2017-08-24 17:25:34",
"入库时间":"2017-08-24 17:25:36"
}
```

图 10-5　入库之前的消息格式

其中，入库时间代表的是车辆数据入库业务消费消息的时间。当将该结构化数据存储在分布式数据库中之后，则通过分析抓拍时间和入库时间就可以从整体上监测数据延时的情况，如图 10-6 所示。

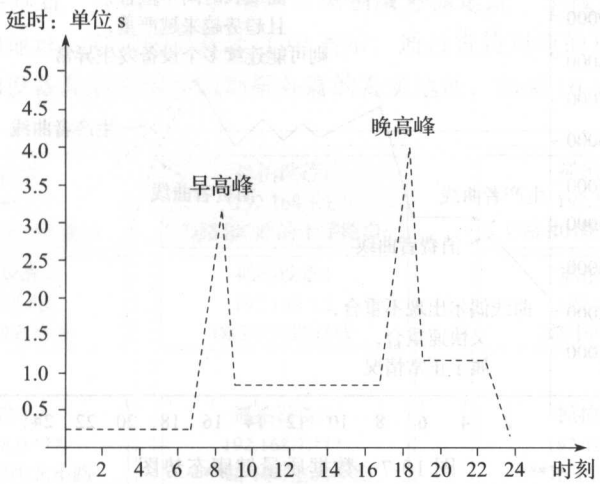


图 10-6　某天的车辆数据延时趋势图

其中横坐标代表一天之内的 24 个时刻，例如 2 代表 02:00，以此类推。纵坐标代表延时，单位为 s。可见某个城市一般在早高峰和晚高峰两个时间段其车辆数据从产生至分布式数据库会产生较大的延时，其他时间段的延时几乎没有。

10.3.4 数据质量的监控

数据质量指的是抓拍设备产生的非结构化数据被解析成结构化数据之后是否可以正常存储在分布式数据库中。比如设备端由于元器件原因导致抓拍的车辆图片（非结构化数据）模糊，肉眼都无法辨认，该图片经过图像识别算法转化为结构化数据，其车牌号码、车牌颜色、车辆颜色、车型类别、车辆品牌、车系、年检标识、纸巾盒、驾驶位是否系安全带和副驾驶位是否系安全带等参数都识别异常。当后端图片识别服务器或者智能抓拍设备将此种类型的结构化数据发送至车辆数据 Topic 之后，车辆数据入库业务消费该 Topic 里面的结构化数据。该业务在把结构化数据存储进分布式数据库之前会剔除异常数据，只会将经过正常识别的结构化数据存储进分布式数据。当生产的结构化数据和正常消费的结构化数据无法一一比对上，则认为此时数据质量是降低的。针对以上情况，业务除了需要监控数据质量之外，还需要知道与产生此异常结构化数据的非结构化数据相关联的普通抓拍设备，以及后端图片识别服务器或者智能抓拍设备，一旦知道相关设备或者服务器之后可以通知运维人员去检修。

那么如何监控数据质量呢？其实就是记录生产者发送的消息个数和消费者正常消费的消息个数，理论上两者应该一致，当出现长时间不一致时，则说明此时可能存在设备损坏的情况，如图 10-7 所示。

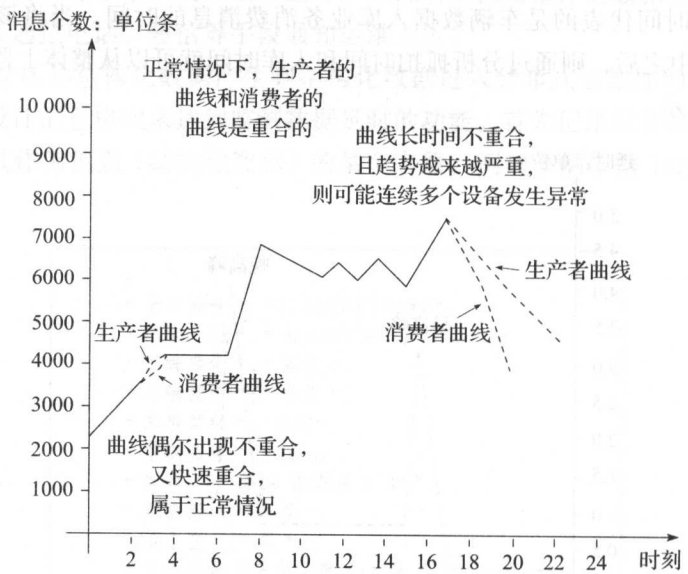


图 10-7 数据质量健康态势图

其中横坐标代表一天之内的 24 个时刻，例如 2 代表 02:00，以此类推。纵坐标代表消息个数，单位是条。正常情况下生产者的曲线和消费者的曲线是重合的，在 03:00 至 04:00 两者曲线出现分叉之后又快速重合，说明设备端出现短暂的异常，例如被树叶挡住，不过当树叶被吹走之后又恢复正常了，但是在 18:00 之后生产者曲线和消费者曲线越来越不重合，

数据质量极速降低，此时说明存在多个设备端陆陆续续出现故障的可能性，需要及时通知检修人员去现场检修。

那么如何知道出现异常的设备端的真实地址呢？由于每个设备端都有 IP 地址，则可以在传递的消息格式中新增代表其来源的设备端 IP 和后端图片识别服务器的 IP，如图 10-8 所示。

```
{
"卡口编号":"110110#1#0#0",
"车牌号码":"识别异常",
"车牌颜色":"识别异常",
"车辆颜色":"识别异常",
"车型类别":"识别异常",
"车辆品牌":"识别异常",
"车系":"识别异常",
"年检标识":"识别异常",
"纸巾盒":"识别异常",
"驾驶位是否系安全带":"识别异常",
"副驾驶位是否系安全带":"识别异常",
......
"抓拍时间":"2017-08-24 17:25:34",
"入库时间":"2017-08-24 17:25:36",
"抓拍设备地址":"192.168.1.1",
"后端图片识别服务器地址":"192.168.1.254"
}
```

图 10-8　新增标识消息来源的消息格式

其中“抓拍设备地址”字段和“后端图片识别服务器地址”字段分别代表抓拍设备和后端图片识别服务的地址，它们的地址通过 IP 表示，通过查找对应的具体项目部署实施图，可以快速定位其抓拍设备和后端图片识别服务器的真实地址，如图 10-9 所示。

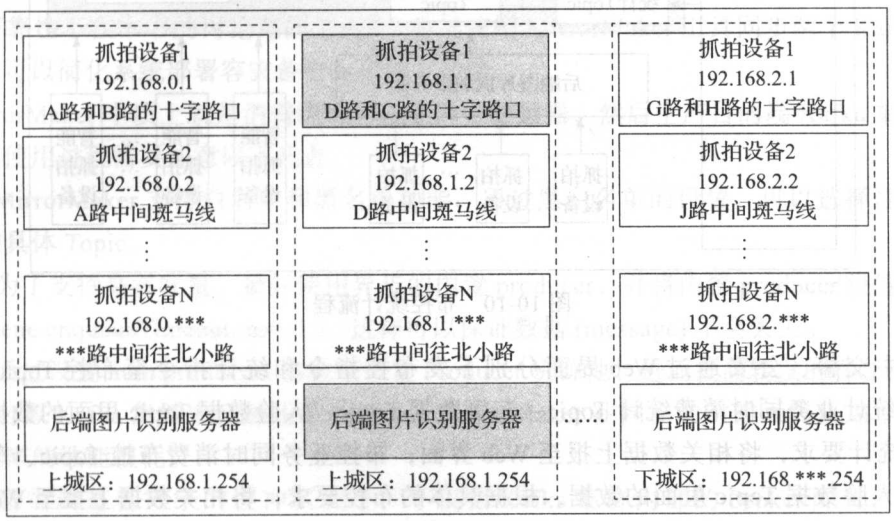


图 10-9　某项目部署实施图

其中后端图片识别服务器地址 192.168.1.254 和抓拍设备地址 192.168.1.1 分别对应上城区的后端图片识别服务器和在该区内的抓拍设备 1，该抓拍设备位于 D 路和 C 路的十字路口。

10.3.5 布控统计

布控指的是实时跟踪某辆车或者某个人的轨迹，统计指的是实时计算某个抓拍设备所负责区域的过车数量或者人流量。在所有典型业务中，布控业务和统计业务实时性要求最高。例如车主向警方举报自己的车牌被他人非法使用，则警方需要在全市布控该车辆，根据车辆的行车轨迹，提前在该车的未来行车轨迹之上布置拦截，有效地当场拦截该车。如果存在分钟级别以上的延时，则无法提前布置拦截。

由于布控统计业务的实时性很高，因此需要尽可能在数据进入分布式数据库之前完成处理。考虑到无法直接利用分布式数据库里面的数据，因此需要直接利用 Kafka 消息队列里面的数据。例如交警将布控指令下发至布控 Topic，布控业务同时消费布控 Topic 里面的消息和车辆数据 Topic 里面的消息，根据具体的布控指令，实时上报被成功匹配的车辆行车轨迹，交警根据车辆行车轨迹提前布置拦截任务，其布控统计的流程如图 10-10 所示。

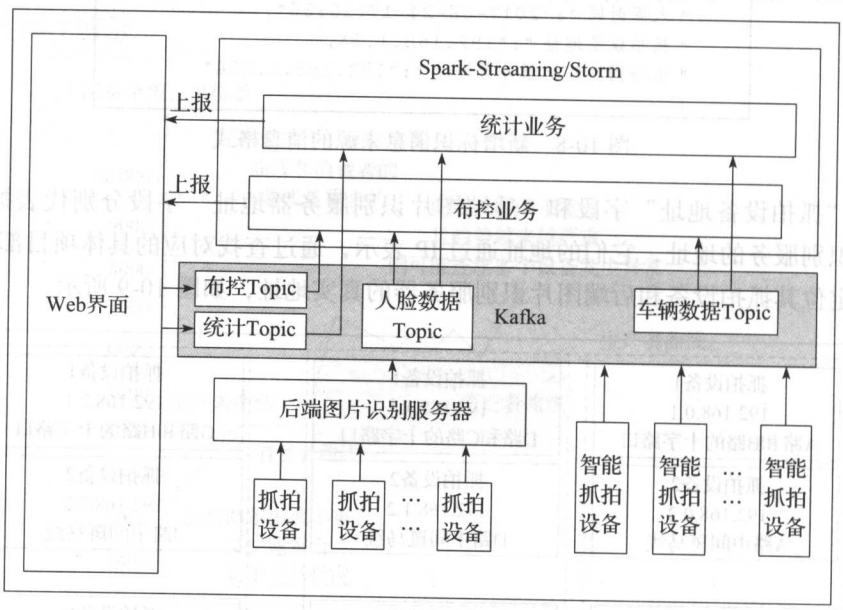


图 10-10 布控统计流程

其中交警、公安通过 Web 界面分别下发布控指令和统计指令至布控 Topic 和统计 Topic。统计业务同时消费统计 Topic、车辆数据 Topic、人脸数据 Topic 里面的数据，根据具体的统计要求，将相关数据上报至 Web 界面；布控业务同时消费布控 Topic、车辆数据 Topic、人脸数据 Topic 里面的数据，根据具体的布控要求，将相关数据上报至 Web 界面。交警、公安通过 Web 界面实时监测统计业务和布控业务上报的实时数据，然后根据具体的

情况做出不同的处理。由于要求高实时性，因此统计业务和布控业务需要基于分布式计算框架中的流式计算引擎 Spark-Streaming 或者 Storm，且此种场景 Storm 比 Spark-Streaming 更合适。那么为什么说 Storm 比 Spark-Streaming 更合适呢？主要基于以下两点：

- Storm 处理的是每次传入的一个事件，而 Spark-Streaming 是处理某个时间段窗口内的事件流。
- Storm 处理一个事件可以达到秒内的延迟，而 Spark-Streaming 处理某个时间段窗口内的事件流则有几秒种的延迟。

10.3.6　容灾备份

容灾备份指的是如果当前部署在公安机房的分布式数据库由于地震、火灾等不可抗力导致数据无法恢复时，如何快速地恢复公安业务，在此场景中数据需要尽可能保持完整，不丢失。传统的做法就是备份当前分布式数据库里面的数据，然后传输至远程存储。当公安机房出现异常无法恢复业务时，需要在远程根据备份数据还原出最近某个时间点的数据，如果两次备份间隔 1 天，则会丢失最多 1 天之内的数据，其他依此类推。此种方式最大的缺点就是丢失数据的时间窗口太长，且丢失的都是最新入库的数据，这样就会对实时性要求高的业务产生较大的影响。考虑到在安防整体解决方案中，Kafka 作为数据总线承担着各个模块数据交换的作用，因此可以在异地部署相同的业务环境，然后通过 Kafka 提供的 MirrorMaker 工具实现两个 Kafka 集群之间数据的同步。如果当前环境出现宕机，则业务可以无缝切换到远程环境，且结构化数据不会出现任何丢失，其高可靠性流程图如图 10-11 所示。

其中 MirrorMaker 从本地业务系统中的 Kafka 集群拉取本地业务系统产生的结构化数据，然后再转发至远程业务系统里面的 Kafka 集群。MirrorMaker 转发结构化数据（数据流），不转发指令类数据（控制流），例如布控 Topic 和统计 Topic 里面的数据（分布式存储里面数据的高可靠性不在本书的讨论范围之内）。通过使用 MirrorMaker 工具同步两个 Kafka 集群的数据，可以简化系统部署容灾备份业务的复杂性。

MirrorMaker 本质上就是消费源 Kafka 集群中的数据，然后生产目的端 Kafka 集群中的数据，其使用过程需要注意以下几点：

1）MirrorMaker 支持白名单和黑名单功能，通过黑白名单的设置，可以选择性地筛选待同步的具体 Topic。

2）为了支持高吞吐量，最好使用异步的内置 producer，并将内置 producer 设置为阻塞模式（queue.enqueueTimeout.ms=-1），这样可以保证数据 (message) 不会丢失。

3）通过设置 num.producers 参数来设置 Producer 资源池，进而提高转发的吞吐量。

4）对源 Kafka 集群中已经压缩的消息，MirrorMaker 在同步的时候无需压缩，避免重复压缩带来的性能损耗。

5）适当调整套接字的缓冲区大小，不仅指 Kafka 集群内部的 Broker 配置，也包括 MirrorMaker 内部的消费者配置。

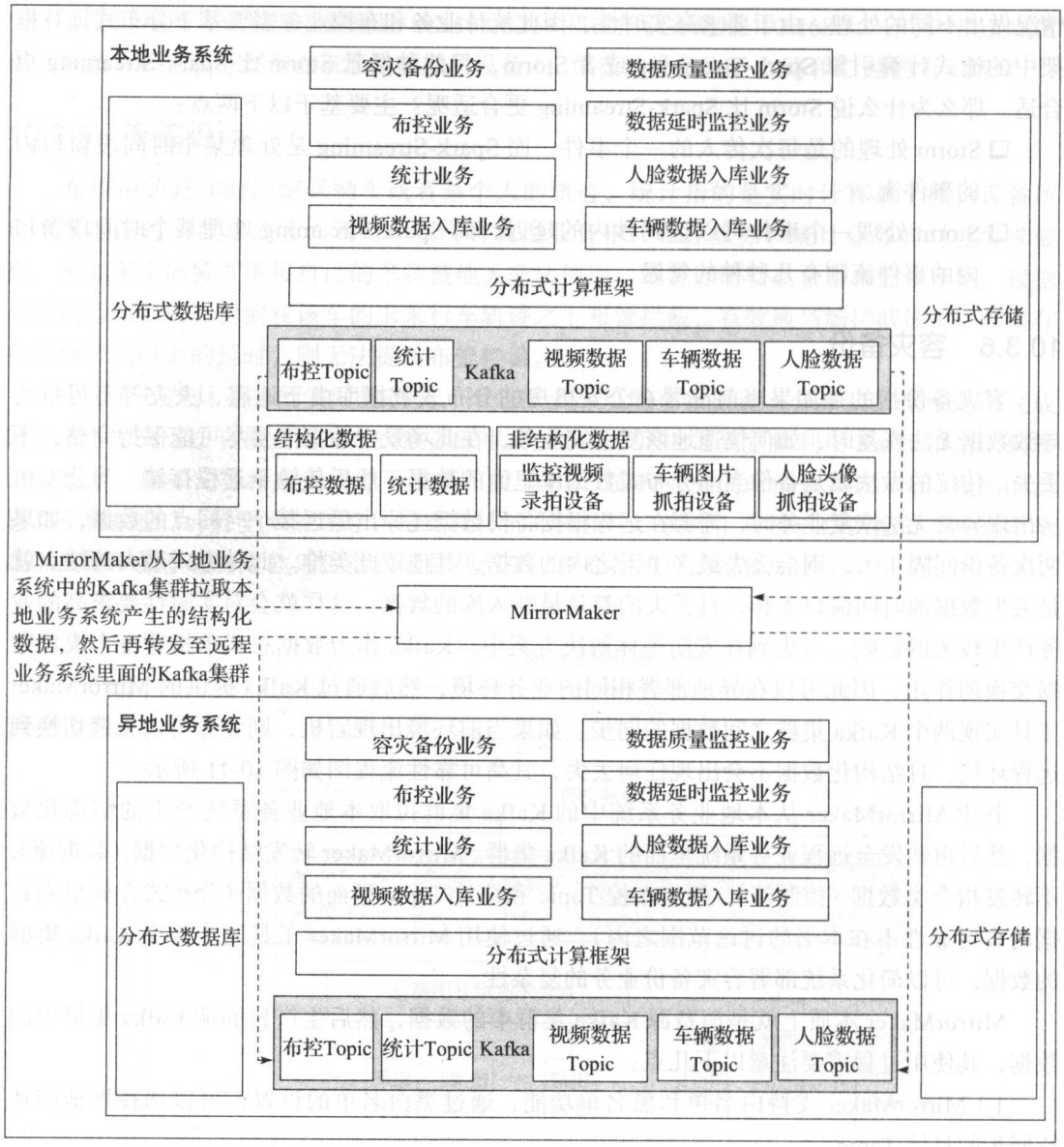
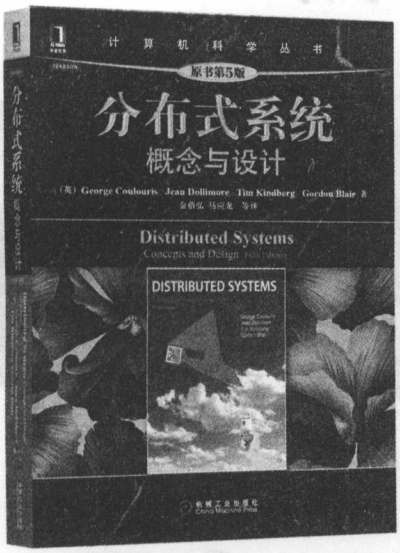


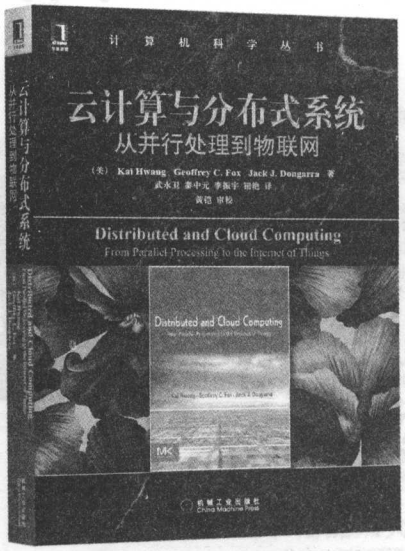
图 10-11 高可靠性流程

10.4 本章小结

本章结合 Kafka 消息队列和安防整体解决方案，简要阐述了 Kafka 作为数据总线在安防系统中的作用。通过理解 Kafka 组件和安防整体解决方案中其他组件的交互，读者可举一反三，深入分析业务数据流，然后将其运用到其他行业的解决方案中。

推荐阅读





分布式系统：概念与设计（原书第5版）

作者：George Coulouris 等 ISBN：978-7-111-40392-0 定价：128.00元

本书全面介绍分布式系统的原理、体系结构、算法和设计，
内容涵盖分布式系统的相关概念、系统模型、数据复制、分布式文件系统、分布式事务、
分布式系统设计等，内容全面，巨细靡遗，是分布式领域的著名教材，被国外多所大学选作为教材。

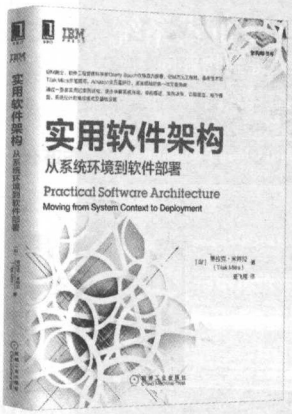
云计算与分布式系统：从并行处理到物联网

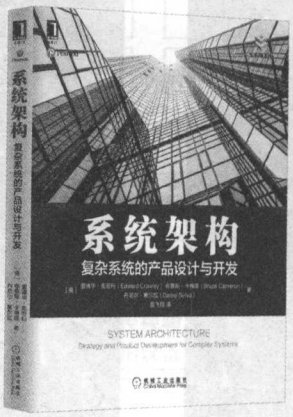
作者：Kai Hwang 等 ISBN：978-7-111-41065-2 定价：85.00元

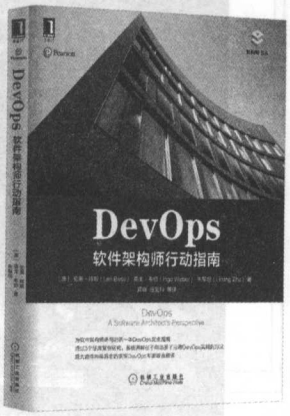
本书覆盖高性能计算、分布式与云计算、虚拟化和网格计算等技术，
阐述了如何为科研、电子商务、社会网络和超级计算等创建高性能、可扩展的可靠系统，
介绍了硬件和软件、系统结构、新的编程范式，以及强调速度性能和节能的生态系统方面的最新进展。
作者将应用与技术趋势相结合，揭示了计算的未来发展，提供的案例研究来自亚马逊、微软、谷歌等。

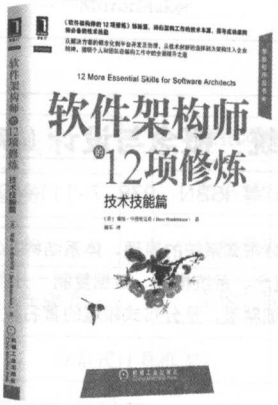
推荐阅读

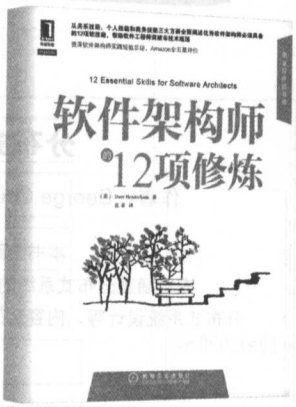
架构师书库











实用软件架构：从系统环境到软件部署

作者：蒂拉克·米特拉 著 ISBN：978-7-111-55026-6 定价：79.00元

系统架构：复杂系统的产品设计与开发

作者：爱德华·克劳利 等 ISBN：978-7-111-55143-0 定价：119.00元

DevOps：软件架构师行动指南

作者：伦恩·拜斯 ISBN：978-7-111-56261-0 定价：69.00元

软件架构

作者：穆拉德·沙巴纳·奥萨拉赫 ISBN：978-7-111-54264-3 定价：59.00元

软件架构师的12项修炼：技术技能篇

作者：戴维·亨德里克森 ISBN：978-7-111-50698-0 定价：59.00元

软件架构师的12项修炼

作者：戴维·亨德里克森 ISBN：978-7-111-37860-0 定价：59.00元

作者简介

王亮

资深架构师，曾在华为担任虚拟化技术工程师，后加入大华公司任分布式数据库系统架构师，研究兴趣为分布式存储、分布式数据库、消息系统等。

本书从LinkedIn（领英）公司内部大数据框架的演变讲起，引申出Kafka消息队列诞生的缘由。接着讲解Kafka的基本架构，围绕Kafka内部的十一条通信协议讲解内部的各个模块的实现细节，包括基本模块和控制管理模块。前者包括SocketServer（监听Socket请求）、KafkaRequestHandlerPool（请求处理资源池）、LogManager（日志管理）、ReplicaManager（分区副本管理）、OffsetManager（偏移量管理）、KafkaScheduler（后台任务调度资源池）、KafkaApis（业务逻辑实现层）、KafkaHealthcheck（提供Broker健康状态）、TopicConfigManager（Topic配置信息管理）；后者包括KafkaController（控制管理模块）。然后介绍Kafka的运维工具，不仅介绍具体如何使用，而且分析其内部的代码实现。紧接着介绍Kafka的客户端编程（生产者和消费者的使用方式），讲解如何进行客户端编程，并分析了生产者和消费者的设计原则和实现原理。最后阐述Kafka和其他开源大数据组件的集成方式以及Kafka在安防整体解决方案中的实际应用。

主要内容包括：

- 为什么要使用消息队列Kafka
- 揭秘典型消息系统内部各基本模块的交互原理
- 典型消息系统元数据管理功能的实现细节
- 运维人员如何正确管理Kafka集群
- 如何编写Kafka客户端代码
- Kafka如何与其他开源大数据组件集成

上架指导：计算机/网络
ISBN 978-7-111-58401-8
定价：79.00元

投稿热线：(010) 88379604
客服热线：(010) 88379426 88361066
购书热线：(010) 68326294 88379649 68995259

华章网站：www.hzbook.com
网上购书：www.china-pub.com
数字阅读：www.hzmedia.com.cn